BUFFER OVERFLOW

INTRODUÇÃO

As vulnerabilidades de buffer overflow são consideradas ameaças críticas de segurança, apesar de ser uma falha bem-conhecida e bastante séria, que se origina exclusivamente na ignorancia do programador referente a aspectos de segurança durante a implementação do programa, o erro repete-se sistematicamente a cada nova versão ou produto liberados.

Este tipo de vulnerabilidade têm sido largamente utilizada para a penetração remota de computadores ligados a uma rede, onde um atacante anônimo tem como objetivo obter acesso ilegal ao computador vulnerável. Mesmo *software* considerado seguro, como o OpenSSH http://www.openssh.org, já apresentou o problema, e também softwares famosos como o Sendmail http://www.sendmail.org> e módulos do Apache http://www.apache.org.

Para se ter uma idéia, das vulnerabilidades já encontradas no ano 2003 e cadastradas no banco de dados http://icat.nist.gov/icat.cfm?function=statistics da ICAT http://icat.nist.gov/, 37% correpondem a *buffer overflow* explorável localmente ou remotamente (num total de 19 falhas). Segundo a mesma fonte, durante o ano de 2002, foram comunicadas 288 falhas também locais ou remotas, totalizando 22% das falhas reportadas naquele ano.

BUFFER OVERFLOWS

Buffer overflows são também chamados de buffer overruns e existem diversos tipos de ataques de estouro de buffer, entre eles stack smashing attacks, ataques contra buffers que se encontram na pilha (vou chamá-la de stack), e heap smashing attacks, que são ataques contra buffers que se encontram na heap. Tecnicamente, um buffer overflow é um problema com a lógica interna do programa, mas a exploração dessa falha pode levar a sérios prejuízos, como por exemplo, o primeiro grande incidente de segurança da Internet - o Morris Worm, em 1988 - utilizava técnicas de estouro de buffer, num programa conhecido como fingerd.

O Objetivo de uma exploração contra um programa privilegiado vulnerável a buffer overflow é conseguir acesso de tal forma que o atacante consiga controlar o programa atacado, e se o programa possuir privilégios suficientes, ou seja se ele possui flag suid root, controlar a máquina.Esses programas com mais privilégios e sem restrições são os programas do usuário root, que é o superusuário de um sistema operacional GNU/Linux, Na maioria das vezes, o atacante tenta subverter as funções de programas que são do superusuário e possuem uma flag suid - que altera o usuário de execução do programa para root, temporariamente - e caso consiga, tenta-se executar um

código arbitrário que dê algum retorno aproveitável, geralmente execução de um interpretador de comandos com nível de superusuário, conhecido como root sh (root shell).

Para que seja possível executar um código arbitrário num programa que está sendo atacado, é necessário utilizar uma das duas técnicas a seguir: injetá-lo ou utilizá-lo, se de alguma forma ele já estiver lá.

Injetá-lo : O programa vítima possui um buffer para armazenar o que será recebido em algum tipo de entrada de dados, a qual é utilizada pelo atacante para fornecer o código arbitrário que será executado (uma string). Na verdade, essa string possui instruções nativas da CPU. Cada byte da string armazena um opcode que é uma instrução válida para a CPU da máquina. Esse conjunto de opcodes é conhecido como shellcode. O buffer, que passou a armazenar o shellcode, pode estar localizado em qualquer espaço de endereçamento do processo: na pilha (variáveis automáticas, também conhecidas como locais), na heap (variáveis alocadas com malloc()) ou na áera de dados static (variáveis inicializadas e não-inicializadas).

Utilizá-lo : dependendo da situação, o código que o atacante deseja executar já se encontra no espaço de endereçamento do programa. Ou seja, o atacante só necessita alterar os parametros de alguma função, de modo que o código desejado seja passado para aquela função e então seja executado.

Organização dos processos em memória

Agora para entendermos como funciona um buffer overflow, nós precisaremos entender como funciona a pilha (stack).
Os processos em execução são divididos em quatro regiões: texto, dados, pilha e *heap.*
A região de texto é fixa pelo programa e inclui as instruções propriamente ditas e os dados somente-leitura. Esta região corresponde ao segmento de texto do binário executável e é normalmente marcada como somente-leitura para que qualquer tentativa de escrevê-la resulte em violação de segmentação (com o objetivo de não permitir código auto-modificável).

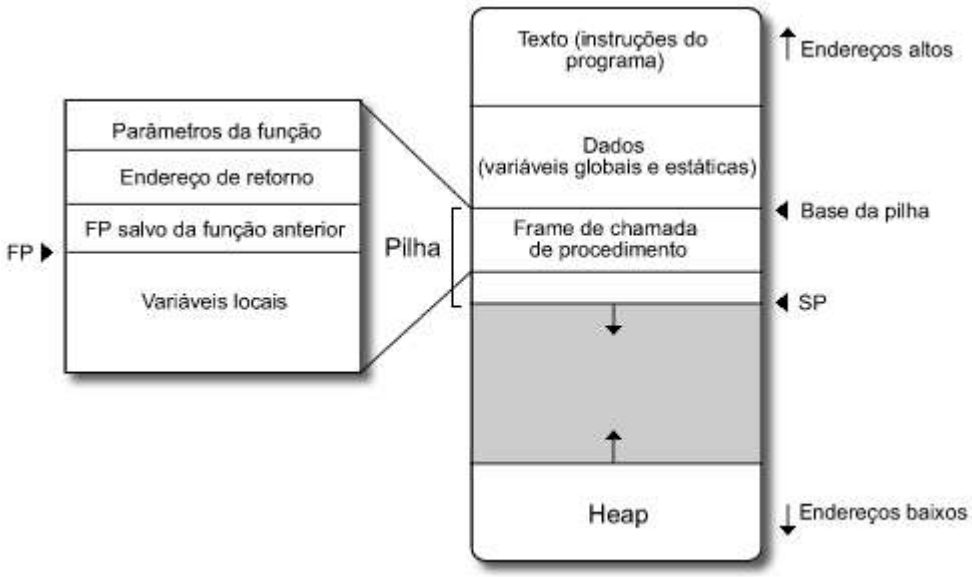
A pilha é um bloco de memória contíguo utilizado para armazenar as variáveis locais, passar parâmetros para funções e armazenar os valores de retornos destas. O endereço de base da pilha é fixo e o acesso à estrutura é realizado por meio das instruções PUSH e POP implementadas pelo processador. O registrador chamado "ponteiro de pilha" (SP) aponta para o topo da pilha.

A pilha consiste em uma seqüência de *frames* que são colocados no topo quando uma função é chamada e retirados ao final da execução. Um *frame* contém os parâmetros para a função, suas variáveis locais, e os dados necessários para recuperar o *frame* anterior, incluindo o valor do ponteiro de instrução no momento da chamada de função.

Dependendo da implementação, a pilha pode crescer em direção aos endereços altos ou baixos. O ponteiro de pilha também é de implementação dependente, podendo apontar para o último endereço ocupado na pilha ou para o próximo endereço livre. Como o texto trata da arquitetura Intel x86, iremos utilizar uma pilha que cresce para os endereços baixos, com o ponteiro de pilha (registrador ESP) apontando para o último endereço da pilha.
Além de um ponteiro de pilha, também é conveniente contar com um "ponteiro de *frame*" (FP) que aponta para um endereço fixo no *frame*. A princípio, variáveis locais podem ser referenciadas fornecendo-se seus deslocamentos em relação ao ponteiro de pilha. Entretanto, quando palavras são inseridas e retiradas da pilha, estes deslocamentos mudam. Apesar de em alguns casos o compilador poder corrigir os deslocamentos observando o número de palavras na pilha, essa gerência é cara. O acesso a variáveis locais a distâncias conhecidas do ponteiro de pilha também iria requerer múltiplas instruções. Desta forma, a marioria dos compiladores utilizam um segundo registrador que aponta para o topo da pilha no início da execução da função, para referenciar tanto variáveis locais como parâmetros, já que suas distâncias não se alteram em relação a este endereço com chamadas a PUSH e POP. Na arquitetura Intel x86, o registrador EBP é utilizado para esse propósito. Por causa da disciplina de crescimento da pilha, parâmetros reais têm deslocamentos positivos e variáveis locais tem deslocamentos negativos a partir de FP.

A primeira instrução que um procedimento deve executar quando chamado é salvar o FP anterior, para que possa ser restaurado ao fim da execução. A função então copia o registrador de ponteiro de pilha para FP para criar o novo ponteiro de *frame* e ajusta o ponteiro de pilha para reservar espaço para as variáveis locais. Este código é chamado de prólogo da função. Ao fim da execução, a pilha deve ser restaurada e a execução deve retomar na instrução seguinte à de chamada da função, o que chamamos de epílogo. As instruções CALL, LEAVE e RET nas máquinas Intel são fornecidas para parte do prólogo e epílogo em chamadas de função. A instrução CALL salva na pilha o endereço da instrução seguinte como endereço de retorno da função chamada. A instrução RET deve ser chamada dentro do procedimento e restaura a execução no endereço que está no topo da pilha.

A *heap* permite a alocação dinâmica de memória por meio de chamadas da família **malloc(3)**. A área de *heap* cresce em sentido oposto à pilha e em direção a esta.



.

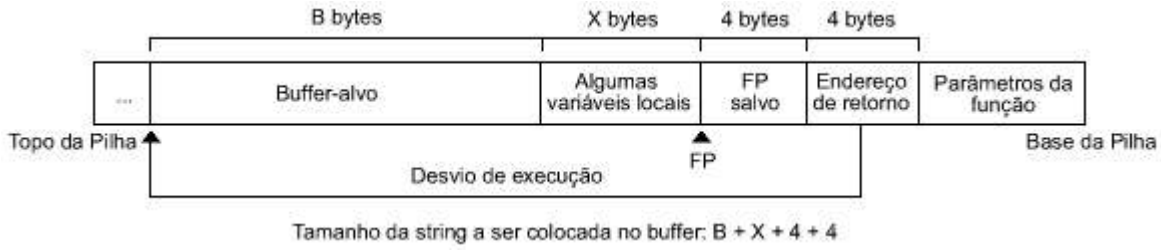
Um *buffer overflow* é resultado do armazenamento em um *buffer* de uma quantidade maior de dados do que sua capacidade . É claro que apenas linguagens de programação que não efetuam checagem de limite ou alteração dinâmica do tamanho do *buffer* são frágeis a este problema.

O princípio é estourar o *buffer* e sobrescrever parte da pilha, alterando o valor das variáveis locais, valores dos parâmetros e/ou o endereço de retorno. Altera-se o endereço de retorno da função para que ele aponte para a área em que o código que se deseja executar encontra-se armazenado (código malicioso dentro do próprio *buffer* estourado ou até algum trecho de código presente no programa vulnerável). Pode-se assim executar código arbitrário com os privilégios do usuário que executa o programa vulnerável. *Daemons* de sistema (**syslogd)**, (**mountd)** ou aplicações que rodam com privilégios de super-usuário (**sendmail)**, até pouco tempo) são portanto alvo preferencial.

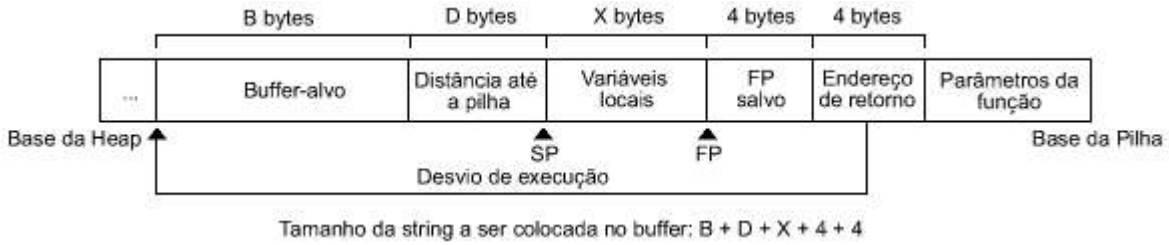
A Tabela 1 lista algumas funções vulneráveis a ataques de buffer overflow da linguagem C.

Existem três tipos básicos de ataques a vulnerabilidades por *buffer overflow*:

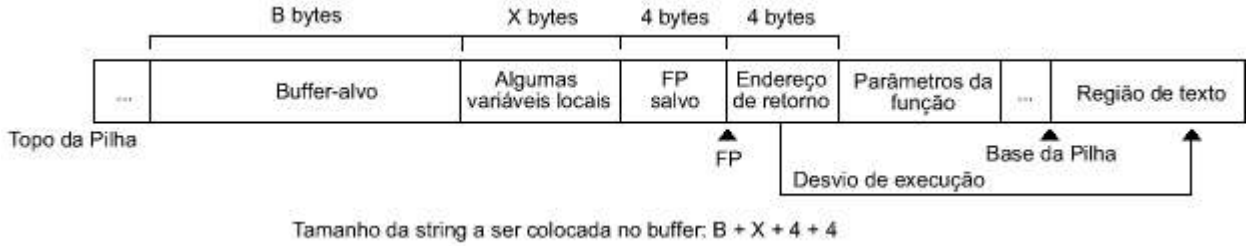
- *Buffer overflow* baseado em pilha: a técnica de exploração mais simples e comum, atua pela alteração do estado da pilha durante a execução do programa para direcionar a execução para o código malicioso contido no *buffer* estourado:



Buffer overflow baseado em *heap*: bem mais difícil de explorar, por causa da disciplina de acesso à *heap* (blocos não contíguos, fragmentação interna). Deve-se estourar o *buffer* armazenado na área da *heap* em direção ao endereço de retorno na pilha, para direcionar a execução para o código malicioso que se encontra no *buffer* estourado;



•*Buffer overflow* de retorno à libc: alteram o fluxo de execução pelo estouro de algum *buffer* na pilha ou *heap*, para algum trecho de código armazenado no segmento de texto do programa. Tipicamente este trecho de código é alguma chamada de função comumente utilizada da biblioteca padrão libc, como as chamadas de execução arbitrária de comandos (funções da família **exec(3)**)..



Exploração de um programa vulnerável

As seções seguintes detalham a exploração de um programa vulnerável a *buffer overflow*, como exemplo de ataque a um programa que apresenta a falha. O programa vulnerável é um servidor TCP, sendo executado em uma máquina Intel, munida do sistema operacional Linux (as distribuições testadas foram a Debian 3.0r1-STABLE e a Gentoo). A idéia geral do ataque é induzir o servidor vulnerável a executar um comando arbitrário a partir de uma chamada à função **exec .** É importante que a chamada de função tenha código pequeno, particularmente a **execve**, porque não se sabe o tamanho do *buffer* a ser estourado.

4.1. Descrição do Programa Vulnerável

Analisaremos agora o trecho de código vulnerável do programa servidor

```
#define BUFFER_SIZE 100

int main(int argc, char *argv[]) {
        int socket_descriptor = -1;
        int incoming_socket;
        char buffer[BUFFER_SIZE];
        int index;
        int message_length;

        while (1) {
                index = 0;
                while ((message_length = read(incoming_socket, buffer + index,
1)) > 0) {
                        index += message_length;
                        if (buffer[index - 1] == '\0')
                                break;
                }
                process(buffer);
        }
}
/* Função de cópia do buffer para processamento externo... */
void process(char *buffer) {
        char local_buffer[100];
        strcpy(local_buffer, buffer);
}
```

O servidor tem duas falhas notáveis, destacadas em vermelho.

O funcionamento básico do servidor resume-se à abertura de um *socket* em modo de escuta para 5 conexões, que recebe uma mensagem de cada cliente conectado, delegando o processamento da *string* recebida à função **process()**. A conexão com um cliente é encerrada quando um *byte* 0 é recebido na mensagem.

A primeira falha diz respeito à chamada da função **read**, que alimenta o *buffer* com *bytes* provenientes do cliente até que seja recebido um *byte* com valor 0. Não há qualquer checagem de limites para o tamanho do *buffer*. Enquanto o cliente enviar *bytes* diferentes de zero, o *buffer* será alimentado, comprometendo possivelmente o estado da pilha. Esta falha não permite a execução de código arbitrário, já que a execução sempre estará presa ao escopo do laço infinito, não sendo possível aproveitar o endereço de retorno que pode ser sobrescrito na pilha.

A segunda falha encontra-se na chamada à função **strcpy** dentro do procedimento **process()**. Assumindo que o *buffer* recebido como argumento foi estourado nas chamadas sucessivas à função **read** sem checagem de limite, a função **strcpy** também estourará o *buffer* local da função durante a cópia da *string*. Isso permitirá a alteração do endereço de retorno armazenado na pilha na chamada à função **process()**, direcionando a execução para qualquer posição de memória. Na exploração

estudada aqui, direcionaremos a execução para o próprio *buffer* recebido como mensagem, que conterá o código malicioso a ser executado.

Foi escolhido o sistema Linux pela simplicidade das chamadas de função da libc (a chamada execve tem em torno de 50 bytes de código binário). Foi realizada uma tentativa com o FreeBSD 5 , mas o tamanho do código das funções de sua biblioteca padrão (cerca de 2,5 KB para a **execve(3)**) tornariam o processo inviável.

5. Técnicas para evitar a vulnerabilidade

A solução tradicional é utilizar funções de biblioteca que não apresentem problemas relacionados a *buffer overflow*. A solução na biblioteca padrão é utilizar as funções **strncpy** e **strncat** que recebem como argumento o número máximo de caracteres copiados entre as *strings.*

Segue abaixo uma tabela com outras opções de funções para evitar este problema:

Função	Risco	Solução
gets()	Extremo	Usar fgets(buffer, tamanho stdin)
Strcpy()	Alto	Usar strncpy() ou strlcpy()
Strcat()	Alto	Usar strncat() ou strlcat()
Sprintf()	Alto	Usar snprintf
Scanf()	Alto	Utilizar especificadores para limitar tamanho
getc()	Moderado	Ao utilizar esta função num loop, verificar buffer destino

fgets()	Baixo	Verificar se o tamanho do destino suporta o argumento da função
snprintf	Baixo	Verificar se o tamanho do destino suporta o argumento da função

As modificações realizadas no sistemas operacional, mais precisamente no kernel do sistema, visam aumentar a proteção do sistema com um todo, e não apenas de uma aplicação isolada. O objetivo destas modificações é tornar o segmento de dados e pilha do espaço de endereçamento de um programa vítima não-executável, fazendo com que seja impossível com que os atacantes executem o código que foi injetado no buffer do programa.