10 de fevereiro de 2011

Entendendo o Buffer Overflow

http://www.crimesciberneticos.com/2011/02/entendendo-o-buffer-overflow.html

Introdução

Estou iniciando uma série de artigos onde pretendo escrever sobre vulnerabilidades, shellcodes, exploits e afins.

É uma área que tenho estudado e descobri ser muito interessante, pois envolve assuntos como assembly, engenharia reversa, programação de baixo-nível, segurança, e que também está no contexto dos crimes cibernéticos.

Por exemplo, uma vulnerabilidade no navegador pode ser explorada quando o usuário acessar um site com códigos-maliciosos, possibilitando assim a instalação de malwares.

Ao invés de utilizar ferramentas prontas resolvi iniciar meus estudos do começo, do nível mais baixo, entender como ocorre uma vulnerabilidade, como é feito um exploit e como é criado o shellcode.

Mesmo que as técnicas estejam um pouco ultrapassadas, acho importante ter o conhecimento da base para depois partir para ferramentas como Metasploit. E para aprender mais nada melhor do que ensinar o pouco que sei.

Então, qual seria a vulnerabilidade mais conhecida e simples de entender?

Buffer Overflow

Frequentemente é noticiado que em uma aplicação qualquer foi encontrada a vulnerabilidade de buffer overflow (ou estouro de buffer) e que através dela um atacante consegue executar código arbitrário. O arbitrário quer dizer qualquer código que ele desejar, ou quase isso.

O comunicado abaixo é um exemplo clássico, emitido pelo CERT em 2003:

> *CA-2003-16 Buffer Overflow in Microsoft RPC*
>
> *A buffer overflow vulnerability exists in Microsoft's Remote Procedure Call (RPC) implementation. A remote attacker could exploit this vulnerability to execute arbitrary code or cause a denial of service.*

Atualmente a vulnerabilidade de buffer overflow é encontrada com menos frequência pois foram criados vários mecanismos de proteção contra ela. Porém ainda ocorre, como podemos ver nessa notícia mais atual, de 27/01/2011:

> *Multiple vulnerabilities in Symantec products*

> *Multiple vulnerabilities have been reported in Symantec products, which can be exploited by malicious people to cause a Denial of Service and compromise a vulnerable system, according to Secunia.*
>
> *1. An error in the Intel AMS2 component when processing certain messages can be exploited to cause a **buffer overflow** via specially crafted packets sent to TCP port 38292.*
> …
> *Successful **exploitation** of the vulnerabilities **may allow execution of arbitrary code**.*
> …
> *The vulnerabilities are reported in the following products: Symantec AntiVirus Corporate EditionServer 10.x and .Symantec System Center 10.x. A solution is to update to version 10.1 MR10.*

Afinal, na prática o que é isso de buffer overflow?

Entendendo o Buffer Overflow

Em programação, buffer é uma variável (também conhecida como array ou vetor), um local na memória que armazena uma quantidade X de bytes.

Por exemplo um buffer que tenha capacidade de armazenar 10 bytes, só conseguiria guardar uma palavra de 9 caracteres (cada caracter sendo 1 byte) já que o último precisa ser o caracter nulo para o programa saber que a palavra termina ali.

Então esse código em C estaria correto:

char buffer[10] = {'S', 'E', 'G', 'U', 'R', 'A', 'N', 'Ç', 'A', '\0'};

Uma variável denominada buffer que tem 10 bytes de capacidade de armazenamento recebe uma palavra de 9 caracteres finalizando com o ('\0'). Isso está correto.

Agora o que aconteceria se eu inserisse uma palavra com mais de 9 caracteres?

Eis o **buffer overflow**! A variável copia somente os 10 primeiros caracteres e o resto estoura, ou transborda, já que não cabe mais nela.

E o resto da sequência após o 10º byte não é descartado, ele sobrescreve o que tiver na memória após a varíavel. Vamos ver um programinha que demonstra isso.

Esse é o código do programa em C:

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

int main(int argc, char *argv[]){
 char buffer1[8] = {'B','U','F','F','E','R','1','\0'};
 char buffer2[8] = {'B','U','F','F','E','R','2','\0'};
```

```
 printf("\n[ANTES] Buffer2 contem: %s\n",buffer2);
 printf("[ANTES] Buffer1 contem: %s\n\n",buffer1);

 strcpy(buffer2,argv[1]);

 printf("[DEPOIS] Buffer2 contem: %s\n",buffer2);
 printf("[DEPOIS] Buffer1 contem: %s\n\n",buffer1);

 return 0;
}
```

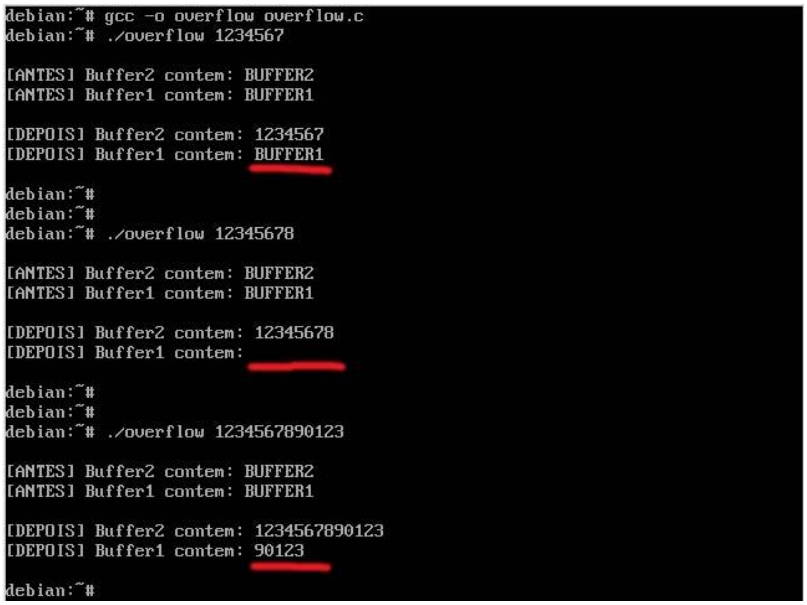
O programa cria uma variável denominada buffer1 com capacidade de armazenamento de 8 bytes e atribui a ela a palavra “BUFFER1” com o caracter nulo finalizando, o mesmo ocorre com a buffer2. Depois exibe o conteúdo de cada uma com o printf.

Na sequência copia para a variável buffer2 o que for passando como argumento na execução do programa e exibe novamente o conteúdo de cada uma.

O programa foi salvo com o nome “overflow.c” e compilado no Linux Debian 3.0 R4 com esse comando:

gcc -o overflow overflow.c

Vejamos algumas execuções do programa com argumentos diferentes:



Na primeira execução foi passada a string “1234567” como argumento para o programa, vemos que ela foi exibida corretamente já que possui 7 bytes e está no limite da capacidade da variável, o programa adiciona o caracter nulo automaticamente ao final da string.

Na segunda execução foi informada a string “12345678” e já vemos aí um buffer overflow. Apesar dela ter 8 bytes (mesma capacidade da variável), ela estourou porque o programa sempre adiciona o nulo ao final. Sendo assim o nulo transbordou o espaço do buffer2 e sobrescreveu o espaço do buffer1.

O mesmo aconteceu na terceira execução, a string foi maior ainda “1234567890123”, dessa vez estourou 5 bytes mais o nulo. A variável buffer1 ficou com esses bytes que passaram da conta.

Detalhe: mesmo a string sendo maior que a variável o programa ainda exibe a string completa no buffer2, isso ocorre porque conforme já mencionado o programa só sabe que um string termina quando ele encontra o nulo.

Caso seja passada como parâmetro uma string maior ainda, o programa irá travar e apresentar um erro de **Segmentation falt**. Isso ocorre porque a string começa sobrescrever vários segmentos de memória que são utilizados para controlar a execução do programa, isso faz com o que o programa “se perca” e trave. Conhece a famosa tela azul do Windows, the blue screen of death? :)

Nosso programa exibindo o erro de segmentation falt:

Justamente a capacidade da variável sobrescrever o que tiver pela frente na memória torna possível tomar o controle do programa e redirecionarmos a execução do mesmo para o que quisermos, isto é, executar código arbitrário. Ao invés de o programa travar podemos fazê-lo executar um shell.

Isso é assunto para o próximo post!

Obs.: os exemplos aqui apresentados podem variar de acordo com a a versão do sistema operacional e do compilador gcc. Utilizei uma versão antiga do Debian, a 3.0 R4, pois nela ainda não havia sido implementadas técnicas de proteção contra buffer overflow. Em posts futuros irei falar sobre isso também.

Adicionado em 15/03/2011:

Para reproduzir os exemplos desse artigo em distribuições Linux mais atuais é necessário desativar algumas proteções, faça o seguinte:

- Debian e Ubuntu based, desativar ASLR:

echo 0 > /proc/sys/kernel/randomize_va_space

- Red Hat based, desativar ASLR e DEP (ExecShield):

echo 0 > /proc/sys/kernel/exec-shield-randomize

echo 0 > /proc/sys/kernel/exec-shield

- GCC a partir da versão 4.1 compilar com diretiva -**fno-stack-protector**, exemplo:

gcc -fno-stack-protector -o overflow overflow.c

Fiz o teste no Debian 5.0.3 com GCC 4.3.2-2 e funcionou corretamente.

Ronaldo Lima
crimesciberneticos.com | twitter.com/crimescibernet

Você também pode gostar de:

- Exploiting Buffer Overflow
- Exploits e Proteções de Memória: ASLR e DEP
- Perícia Forense Computacional: metodologia
- Grupo sobre Exploits e Vulnerabilidades
- Sorteio do livro: Desvendando a Computação Forense