Uma metodologia para estudo de vulnerabilidades de *BUFFER OVERFLOW*

Francilene Coelho Cavalcante[1], Claudio de Castro Monteiro[2]
[1]Doutorandos do Programa de Pós-Graduação em Telemática – IFTO. e-mail: fulanodetal@ifto.edu.br
prof doutor do Isnt...
[2]Mestrandos do Programa de Pós-Graduação em Telemática - IFTO. Bolsistas do CNPq. e-mail: fulaninhosdetal@ifto.edu.br

Resumo: Este trabalho é um estudo sobre buffer overflow, na qual são apresentadas as arquiteturas de softwares que apresentam o problema. O ambiente utilizado para exploração, arquiteturas, softwares, bem como a própria exploração são detalhados durante esse trabalho. O problema de buffer overflow é basicamente estouro de memória, onde os dados transbordados sobrescrevem áreas adjacentes de memória. Comenta-se ainda alguns métodos adotados pelos próprios sistemas operacionais como medida de segurança para este tipo de falha, bem como os cuidados devem ser tomados durante a programação de um sistema. Nos experimentos, foi possível invadir sistemas de arquiteturas x86 e x86_64, local e remoto.

Palavras–chave: Arquitetura, *buffer*, falha, operacional, *overflow*, programação, segurança, sistema

1. INTRODUÇÃO

Buffer é uma área temporária da memória onde ficam guardados dados para serem manipulados. *Overflow* é um transbordamento, e ocorre quando o volume de um determinado objeto é maior do que o seu tamanho. *Buffer overflow* consiste em armazenar em um *buffer* de tamanho fixo, dados maiores que o seu tamanho, causando seu estouro, caracterizando esta falha como um erro de programação.

O objetivo de estourar a pilha, é poder sobrescrevê-la, alterando o valor das variáveis locais, valores de parâmetros ou endereço de retorno. Altera-se o endereço de retorno da função para que ele aponte para área em que se deseja executar o código malicioso dentro do *buffer* estourado.

Algumas linguagens de programação tornam um sistema exposto a esta vulnerabilidade. C, por exemplo, é uma linguagem que não verifica automaticamente se o espaço reservado para uma variável está dentro de seus limites, deixando esta função ao programador, que deve sempre estar atento.

Os códigos maliciosos, também conhecidos como *exploits*, são programas que se aproveitam de vulnerabilidades, visando causar comportamento errôneo, inesperado ou até ganhar privilégios de superusuário em sistemas.

Este trabalho apresenta o principio para explorar vulnerabilidade de *buffer overflow.* Além de explorar esta vulnerabilidade, este trabalho visa alertar aos desenvolvedores sobre os danos causados por um simples erro de programação. O trabalho está organizado em seções: mostrando como um processo se organiza na pilha (seção 2), os registradores importantes para manuseio da mesma (seção 2.1), o principio de um ataque local e remoto (seção 3), conceitos e utilização de *shellcode* (seção 4), e as medidas de segurança adotadas para se precaver deste tipo de ataque (seção 5).

2. REFERENCIAL TEÓRICO

Vários projetos tem abordado o problema de *buffer overflow*. Etoh, Hiroaki. (2000), apresenta ideia para melhorar na detecção de estouro de buffer.

Cowan, C., Wagle, P. et al .(2003) apresenta formas de prevenção, chamado *Canary* (Canário), quando uma chamada de função é feita, um "canário" é adicionado ao endereço do remetente, se ocorrer o *buffer overflow*, o canário será corrompido. *Stack Guard* usa essa técnica, implementando-o como um *patch* para o compilador GCC, o que provoca atrasos mínimos de desempenho.

ISBN 978-85-62830-10-5

2.1. ORGANIZAÇÃO DA PILHA

Uma pilha (*stack)* adota um padrão chamado LIFO (*Last In First Out*), o último que entra, é o primeiro a sair. Ela é um local reservado na memória RAM, onde o programa armazena variáveis locais de uma função, e controla a execução de um programa. À medida que uma sub-rotina é chamada, os dados são empilhados, sendo desempilhados quando a mesma termina. Um espaço específico da pilha é chamado de *stack frame.*

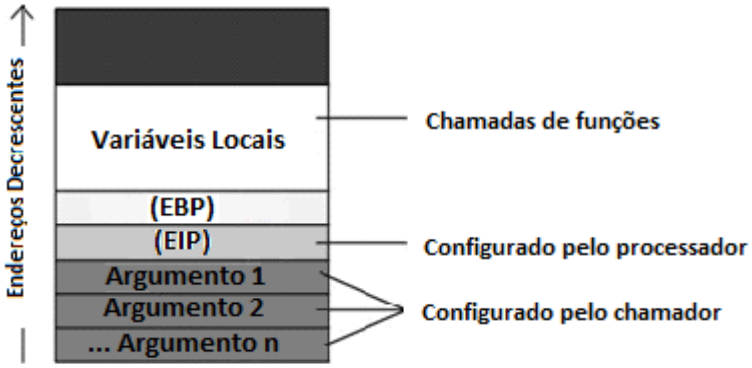

Figura 1- Estrutura da pilha

Cada *frame*, no momento da chamada de uma função, possui dados como variáveis locais, parâmetros da função, endereço de retorno para o *frame* anterior e o valor do ponteiro de instrução (EIP, ou *Extended Instruction Pointer*). Um *frame* é colocado na pilha na chamada de uma função, e retirado no seu retorno.

As pilhas possuem basicamente duas operações: *PUSH* que insere os dados, e *POP* que retira os dados da memória. Para controlar as operações de inserção e remoção da pilha, é usado o registrador *stack pointer* (SP*)*, que aponta para o seu topo. Ao executar o comando *PUSH*, um valor é inserido na pilha e o registrador é decrementado. Na chamada do comando POP, o valor do qual o *stack pointer* aponta é retirado da pilha, e seu valor é incrementado (ALEFHONE, 1996).

Para acessar variáveis locais e parâmetros da função, é usado um segundo registrador chamado *frame pointer* (FP), que aponta para um endereço fixo na pilha.

2.2. REGISTRADORES

Registradores são locais no processador para armazenar dados temporariamente. Os registradores de uso geral da arquitetura x86 são:

Os registradores da arquitetura x86:	**Os registradores arquitetura x86-64 são:**
EAX-registrador acumulador	RAX - registrador valor de retorno
EBX - registrador base	RBX - registrador base
ECX - registrador contador	RCX - registrador contador
EDX - registrador de dados	RDX - registrador de dados
ESI - registrador de índice da fonte dos dados	RSI - registrador de índice da fonte dos dados

ISBN 978-85-62830-10-5

EDI - registrador de índice do destino dos dados	RDI - registrador de índice do destino dos dados
EBP - registrador ponteiro para a moldura de chamada de função	RBP - registrador ponteiro para a moldura de chamada de função
ESP - registrador ponteiro para a pilha de execução	RSP - registrador ponteiro para a pilha de execução
EIP - registrador de ponteiro de instrução (registrador de uso específico).	R8 - registrador de dados
	R9 - registrador de dados
	R10 - registrador ponteiro para a moldura de chamada de função
	R11 - registrador de *linking*
	R12, R13, R14, R15 - registradores de base.
	RIP - registrador de ponteiro de instrução (registrador de uso específico).

Dentro das arquiteturas, x86 ou x86_64, os registradores que merecem atenção para explorar a vulnerabilidade de *buffer overflow* são: ESP, EBP e EIP, RSP, RBP e RIP, respectivamente. O ESP aponta para o topo da pilha, e seu valor pode ser modificado. O EBP contém o endereço do fundo da pilha. A cada chamada de função, o registro de EBP é o primeiro a ser sobrescrito na pilha e, em seguida, o novo valor do ESP é movido para EBP. Uma vez que ESP aponta para o topo da pilha, ele é alterado com frequência durante a execução de um programa. O registrador de ponteiro de instrução (EIP - RIP) possui o endereço para que, após a execução a uma chamada de função, o programa possa retornar para a função chamadora. Esse registrador é o foco para a exploração. É ele que guarda o endereço de retorno das funções, e uma vez alcançado esse endereço, pode-se então mudar o retorno dessa execução, tomando assim controle do programa.

Para seguir a exploração de *buffer overflow* é necessário ter o programa vulnerável. A partir disso estoura a pilha, de forma que os dados estoure e transbordem, sobrescrevendo áreas adjacentes de memória. Com um depurador, chamado gdb, faz-se a analise da pilha para então poder inserir o *shellcode*, explicado na seção 4. Feito isso, controla-se os registradores EIP/RIP, tomando controle do sistema.

2.3. BUFFER OVERFLOW LOCAL E REMOTO

Existem basicamente três tipos de exploração de *buffer overflow*: pilha, *libc e heap*. A exploração utilizada para este trabalho é baseado em pilha. Devido sua simplicidade e sua grande utilização em explorações.

Quando uma falha de segurança é explorada, pode causar sérios danos, dependendo de quem ou de como se pretende utilizar as informações adquiridas durante o processo de exploração.

O fato é que, erros aparentemente simples, podem resultar em consequências graves. Um exemplo disto é o *buffer overflow*.

Sistemas vulneráveis sejam locais ou remotos, quando explorados, causam o mesmo impacto. A diferença é que no ataque local, o invasor deve ter acesso à máquina para que o ataque seja efetuado.

O ataque remoto de *buffer overflow* acontece quando o invasor explora a vulnerabilidade em uma máquina onde seu acesso é dado por uma infraestrutura de comunicação (rede de computadores).

ISBN 978-85-62830-10-5

O código a seguir é um exemplo de programa vulnerável. É utilizado o mesmo código para exploração nas arquiteturas x86 e x86_64 em distribuições Linux, como principio da exploração de *buffer overflow*. O programa está vulnerável porque ao utilizar a função da biblioteca da linguagem C: *strcpy*, tudo que é passado como parâmetro é copiado para dentro da *string* declarada 'variavel'. Funções além dessa, como ***gets (), fgets (),*** dentre outras, ao serem utilizadas deve-se ter o cuidado para validar os dados de entrada. Assim evita-se estar vulnerável à *buffer overflow.*

2.4. SHELLCODE

Para manipular diretamente os registradores e funções de um programa, é necessário um *shellcode*. Que pode ser definido segundo (ANLEY, 2007), como um conjunto de instruções que são injetados e executados por um programa através de um *exploit*. Normalmente, é escrito em *assembly* por manipular diretamente os registradores e depois transformado em *opcodes* em hexadecimal.

O *shellcode* é colocado em uma área especifica da memória, e então o programa é levado a executar o *shellcode* fornecido.

Como há diferenças no endereçamento de memória nas arquiteturas x86 e x86_64, para cada arquitetura há um *shellcode* especifico a ser utilizado. Mas ambos tem a mesma função de mudar o endereço de retorno do registrador (EIP, RIP), respectivamente x86 e x86_64, o fazendo apontar para o código malicioso, onde me retornará um *Shell* com permissões de um superusuário.

3. RESULTADOS

3.1. EXPLORAÇÃO

A base de exploração é a mesma para os ataques local e remoto, arquiteturas 32 e 64 bits. A maior diferença é localizar a posição exata de memória para passar como instrução, e assim obter sucesso na exploração.

Foram feitos testes para cada tipo de exploração. Para explorar *buffer overflow* local e remoto em arquitetura 32 bits, foram feitos testes com sistema operacional Ubuntu 11.10, maquinas de uso pessoal, processador Intel core i5, Intel core 2 duo, kernel 3.0.0-22-generic.

Para explorar a vulnerabilidade da arquitetura x86_64, foram utilizados maquinas de laboratório da própria faculdade, o ambiente para teste foi um sistema operacional Fedora 15 com a versão de kernel 2.6.38-26.rc1.fc15.x86_64 e processador AMD Phenom (tm) II X2 550. Sistemas operacionais de 64 bits são capazes de mapear mais áreas de memória que os sistemas de 32 bits, fazendo com que os registradores trabalhem com quantidades maiores de bytes.

A sequencia de imagens que seguirão, descrevem os passos para se explorar a vulnerabilidade de *buffer overflow*.

3.2. LOCAL

Como foi dito na sessão 2.3, a exploração para este trabalho é baseada em pilha. Serão mostrados através de imagens, os passos seguidos para explorar *buffer overflow*.

Figura 2- Compilando o programa.

O programa deve ser compilado conforme mostra a figura 2. Na compilação do *buffer* local x86, o que difere é o parâmetro passado em *–mpreferred-stack-boundary*=4, onde o numero 4 indica como a pilha estará se organizando. E que deverá ser trocada para 2, quando o programa for compilado para explorar na arquitetura x86. Pois a diferença entre as duas arquiteturas é a forma de como os registradores trabalham e como os sistema operacional mapea suas áreas de memória.

ISBN 978-85-62830-10-5

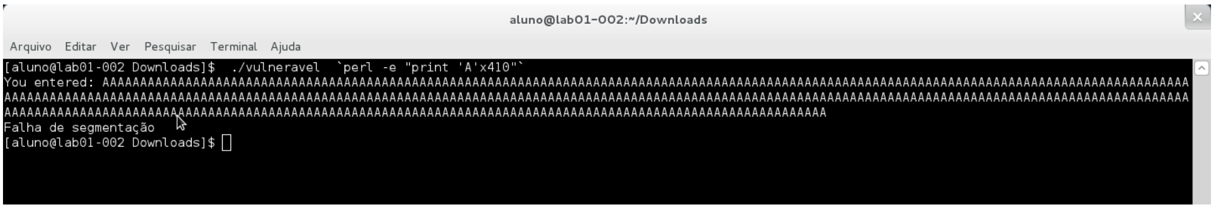

Figura 3-Estouro de buffer

O código acima é uma primeira tentativa de estourar o *buffer* de memória, causando a falha de segmentação. O comando "*perl*" é usado para diminuir a escrita de código, para não ter que escrever 410 'A', já que se deve passar valores maiores que uma variável está predefinida a receber. Para avaliar o funcionamento da pilha, executa o mesmo comando dentro do depurador chamado *gdb*.

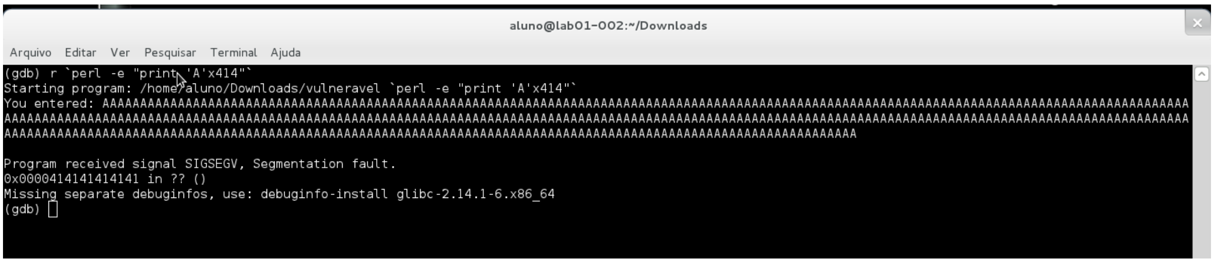

Figura 4- Estouro de buffer dentro do depurador gdb

Os valores passados como parâmetros mudam de uma arquitetura para outra, 414 é o valor exato de "A" para sobrescrever a pilha. Numa arquitetura 32 bits os valores devem ser menores, pois a pilha foi organizada de quatro em quatro bytes, definidos na figura 2. Sendo que para alcançar esse retorno mostrado, foram realizados tentativas com vários outros valores, até chegar a esse resultado.

Uma vez que a pilha já está sobrescrita, é necessário inserir o *shellcode*, detalhado na sessão 2.4. Para cada arquitetura há um tipo especifico de *shellcode*, mostrado nas próximas imagens:

Figura 5- Shellcode x86_64

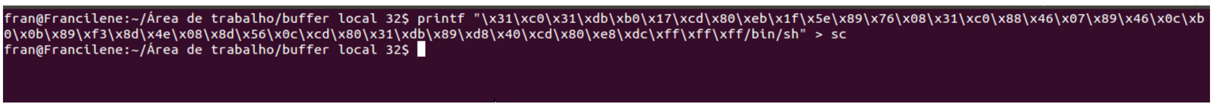

Figura 6- Shellcode x86

Figura 7- Inserindo shellcode na pilha

ISBN 978-85-62830-10-5
VII CONNEPI©2012

O *shellcode* inserido, faz com nos retorne um *Shell* com privilégios de superusuário. Pessoas com conhecimentos em *assembly* podem construir seu próprio *shellcode*. Podendo fazer com que ao ser inserido manipule o sistema, mudando instruções, chamadas de funções e assim tomar controle do sistema.

Lembrando ainda que todos os valores mostrados nas figuras são exatos, e que foram realizados vários testes com valores diferentes.

Na arquitetura x86, foi passado o seguinte comando como parâmetro dentro do depurador: *r `perl –e 'print "A"x351' ``cat sc`` perl –e "A"x4.*Ainda dentro do depurador devemos localizar os endereços de memória onde estão inseridos os "A".

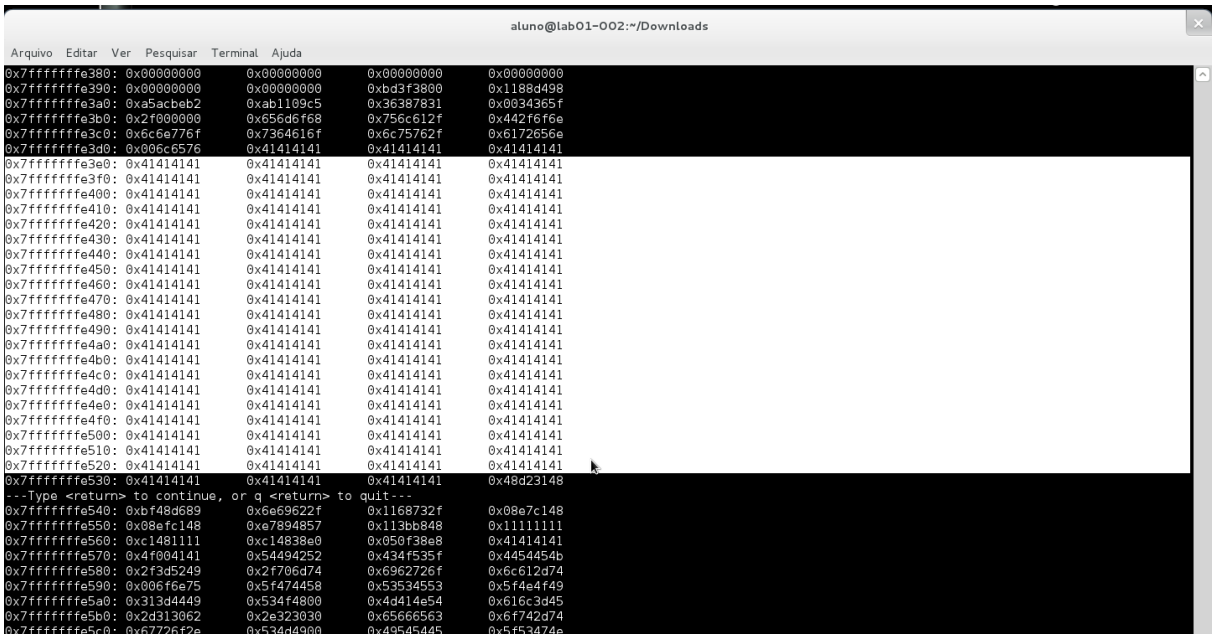

Figura 8- Localizando endereços de memória.

Para fazer essa localização na arquitetura x86_64, foi utilizado o comando *x/400x $rsp*. No caso da arquitetura x86, deve-se trocar o *$rsp* por *$esp*, que é o registrador responsável pelo retorno de funções.

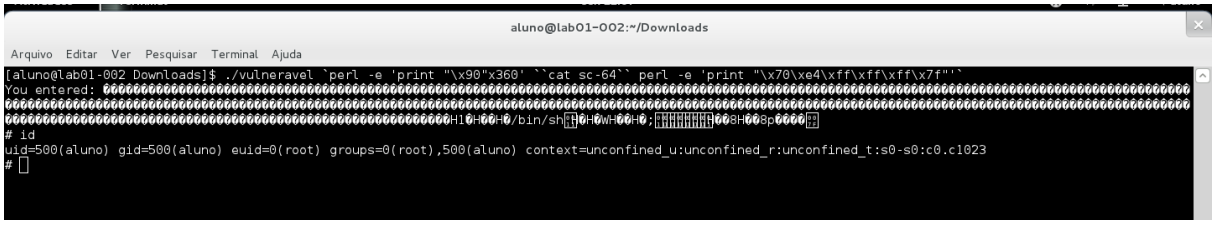

Figura 9- Shell retornado.

Figura 10 - Shell retornado em arquitetura x86

ISBN 978-85-62830-10-5

Depois de localizado os endereços, agora é só pegar um endereço especifico aqui escolhido o 0x7fffffffe470 para x86_64 e 0xbffff430 para x86 e passar como parâmetro. Assim um *Shell* é retornado com os privilégios de um superusuário.

3.3. REMOTO

A exploração remota segue a mesma linha de raciocínio de uma exploração local. A única diferença é a comunicação entre as maquinas, que se deve através de uma rede de computador e a utilização da programação usando sockets.

As compilações dos programas seguem a mesma forma mostrada na figura 2. No entanto o inicializa o servidor e em uma maquina cliente passa os parâmetros.

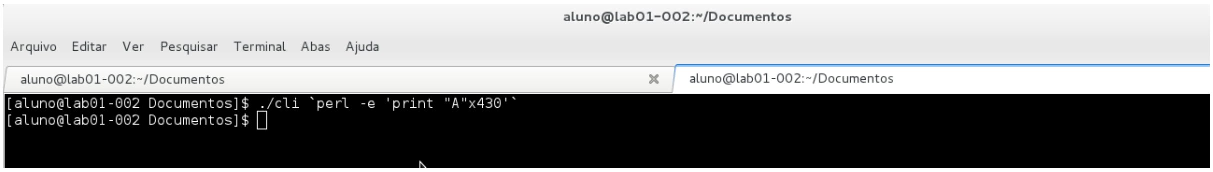

Figura 11 - Tentativa de estouro de buffer

O comando usado na arquitetura x86 é o mesmo usado na figura 11, mudando somente o valor passado, que deve ser menor a este. O resultado do comando é mostrado na figura 12, onde a pilha é sobrescrita.

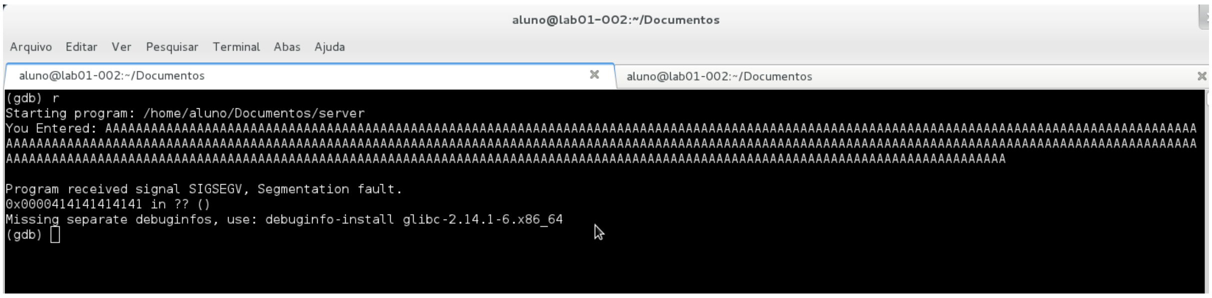

Figura 12 - Estouro de buffer dentro do depurador.

O shellcode é o mesmo utilizado e mostrado nas figuras 5 e 6. Inicializando o servidor, e com o comando: ./cli `perl –e 'print "A"x376' ``cat sc-64 `` perl –e 'print "A"x6'` passado pelo cliente, é possível sobrescrever o endereço de retorno da pilha. Na arquitetura x86 troca-se 376 por 355, e o 6 por 4. Localizando os endereços de memória como foi feito na figura 8, agora é passar como parâmetro.

ISBN 978-85-62830-10-5

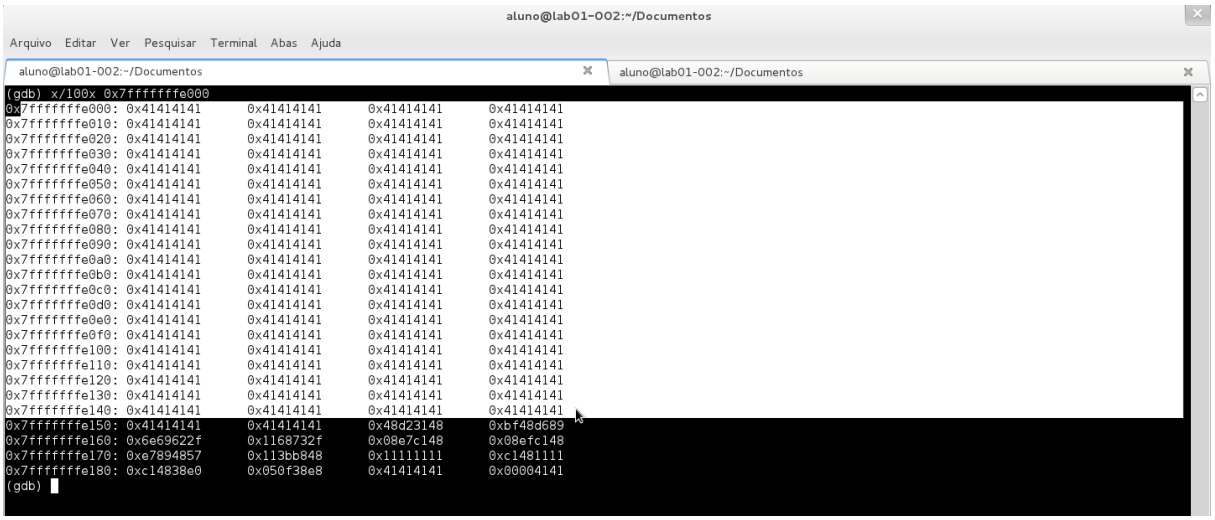

Figura 13 - localizando endereços de memória.

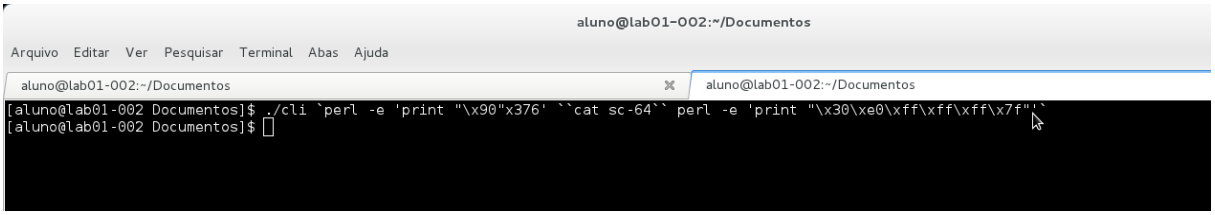

Figura 14 - Passando endereço de memória

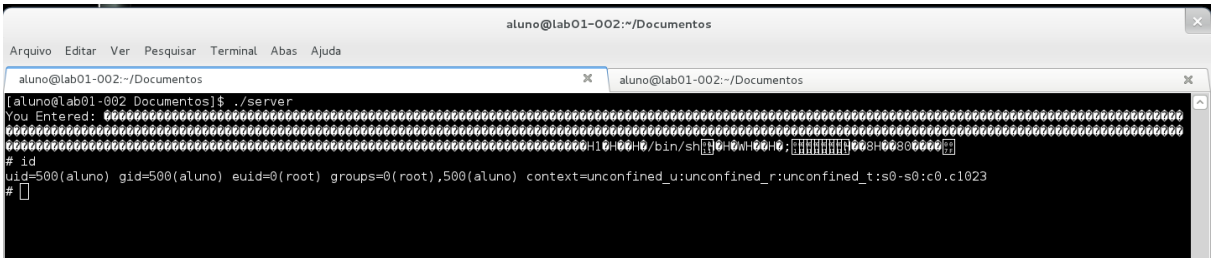

Figura 15 - Shell de superusuário.

As figuras 14 e 15 são semelhantes à figura 9. Onde o cliente passa o endereço 0x7fffffffe030 para o servidor, obtendo o *Shell* de *root*. O endereço usado para a arquitetura x86 foi o 0xbffff241, como pode ser observado na figura 16.

Figura 16 - Passagem de endereço.

4. MEDIDAS DE SEGURANÇA

Diversos métodos surgiram a fim de sanar o problema de *buffer overflow*. A randomização do *kernel* é um exemplo disso, que é uma medida de segurança adotada para que cada vez que um

ISBN 978-85-62830-10-5

processo seja executado, a pilha de processos se movimente de forma aleatória, dificultando a descoberta da posição exata do topo da pilha e do endereço de retorno das funções.

Dentre as técnicas existentes e usadas em compiladores para proteção de código escrito em C, destaca-se o *Stack-Smashing Protector* (ou SSP, ou ainda Pro Police) (ETOH,2000). Ele é uma evolução do conceito apresentado no Stack Guard (COWAN, 2003). Atualmente, o método é padrão em diversos sistemas operacionais conhecidos, como *Ubuntu* e *OpenBSD.*

5. CONCLUSÃO

Este trabalha apresentou uma visão geral sobre buffer overflow. Foi mostrado o roteiro de como se explorar essa vulnerabilidade em arquiteturas de 32 e 64 bits, local e remoto. Ainda que uma falha resultante de um erro de programação, os sistemas tentam se precaver, dificultando o acesso do endereço exato dos registradores necessários para manipular e tomar controle do sistema.

Além das medidas de segurança adotadas pelo sistema, o problema de buffer overflow pode ser evitado com boas praticas de programação. Aumentando, assim, a dificuldade de exploração e controle do sistema.

REFERÊNCIAS

Cowan, C., Wagle, P. et al .(2003). ***Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.*** DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00.

Etoh, Hiroaki. (2000). ***Protecting from stack-smashing attacks.*** http://www.research.ibm.com/trl/projects/security/ssp/main.html

ALEPHONE, **Smashing The Stack For Fun and Profit**. Volume 7, edição 49. Novembro de 1996. www.phrack.com/issues.html?issue=49&id=14

ANLEY, C. **The Shellcoder's Handbook**: discovering and exploring security holes. 2ª. Ed. [S.I]: Wiley Publishing, Inc., 2007.

ISBN 978-85-62830-10-5