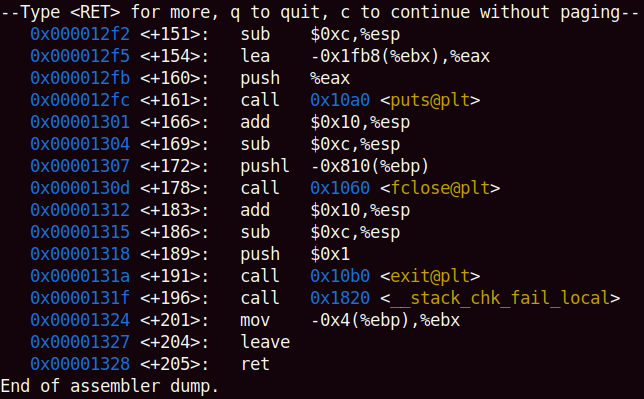
Laboratório 2 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

**Função Inicializa:**

Antes de chamar as funções de cada bomba, a função main chama uma outra, chamada inicializa. Ela simplesmente espera um argumento a ser passado após o ./bomb (primeiraSenha, segundaSenha...). Caso nenhum argumento seja passado, ela detona a bomba, colocando puts(“Explodiu!”) e fecha o arquivo em seguida com exit. Isso também se aplica para o comando run no GDB, que usamos após um breakpoint para poder explorar melhor o código.



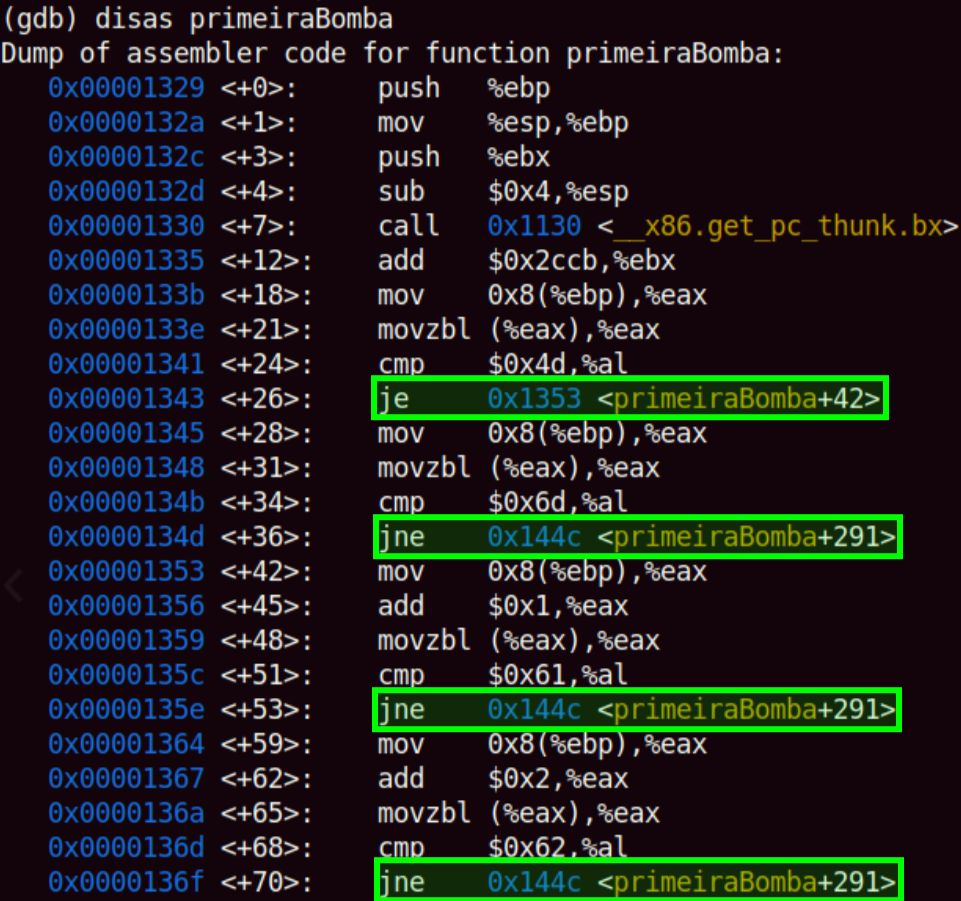
Acima está o trecho do código responsável por fechar o arquivo caso não digitemos uma entrada.

**Primeira Bomba:**

O primeiro desafio foi o mais fácil dos quatro. Ao digitarmos o comando disas primeiraBomba, logo nos atentamos para a grande quantidade de instruções cmp que o código tinha. Os operandos dessas instruções eram sempre um número em hexadecimal comparado com o byte menos significativo do registrador %eax, que é %al. Tivemos a ideia de verificar se esses hexadecimais tinham códigos correspondentes na tabela ASCII (e tinham). Depois disso só precisamos verificar o restante dos códigos na tabela para decifrar a primeira senha

Mab – 353 – 2020.2 ou mab – 353 – 2020.2

Inicialmente, há uma verificação se estamos digitando o caractere correspondente a 0x4D, que é a letra “M”. Caso sim, ocorre um desvio, que salta a verificação da letra “m” (0x6D) direto para a da letra “a” (0x61). A partir daí, em todas as verificações há um jne 0x144C <primeiraBomba+291>, que detona a bomba chamando a função boom caso o caractere lido seja diferente do esperado.



Destacados na imagem acima estão, respectivamente:

* Verificação de “M” e possível salto para verificação de “a”
* Verificação de “m” e possível salto para explosão
* Verificação de “a” e possível salto para explosão
* Verificação de “b” e possível salto para explosão

**Segunda Bomba:**

A segunda bomba foi consideravelmente mais difícil que a primeira. Inicialmente, notamos que não há instruções cmp no código e, por isso, tivemos de pensar em outra maneira para descobrir a senha. Percebemos depois, que novamente existia uma grande quantidade de instruções de mesmo nome (movl). Três delas destoavam do restante, pois copiavam valores diferentes de zero para posições da pilha, enquanto as outras copiavam o valor zero. Ao analisar os operandos dessas instruções, percebemos que haviam sempre pares de números em hexadecimal sendo copiados, o que mais uma vez nos fez pensar na tabela ASCII. Verificamos os caracteres correspondentes dos seguintes números:

0x20 0x62 0x61 0x4C 🡪 0x42 0x20 0x3A 0x32 🡪 0x62 0x6D 0x6F

A partir disso, obtivemos a seguinte sequência de caracteres: [espaço]baL B :2 bmo

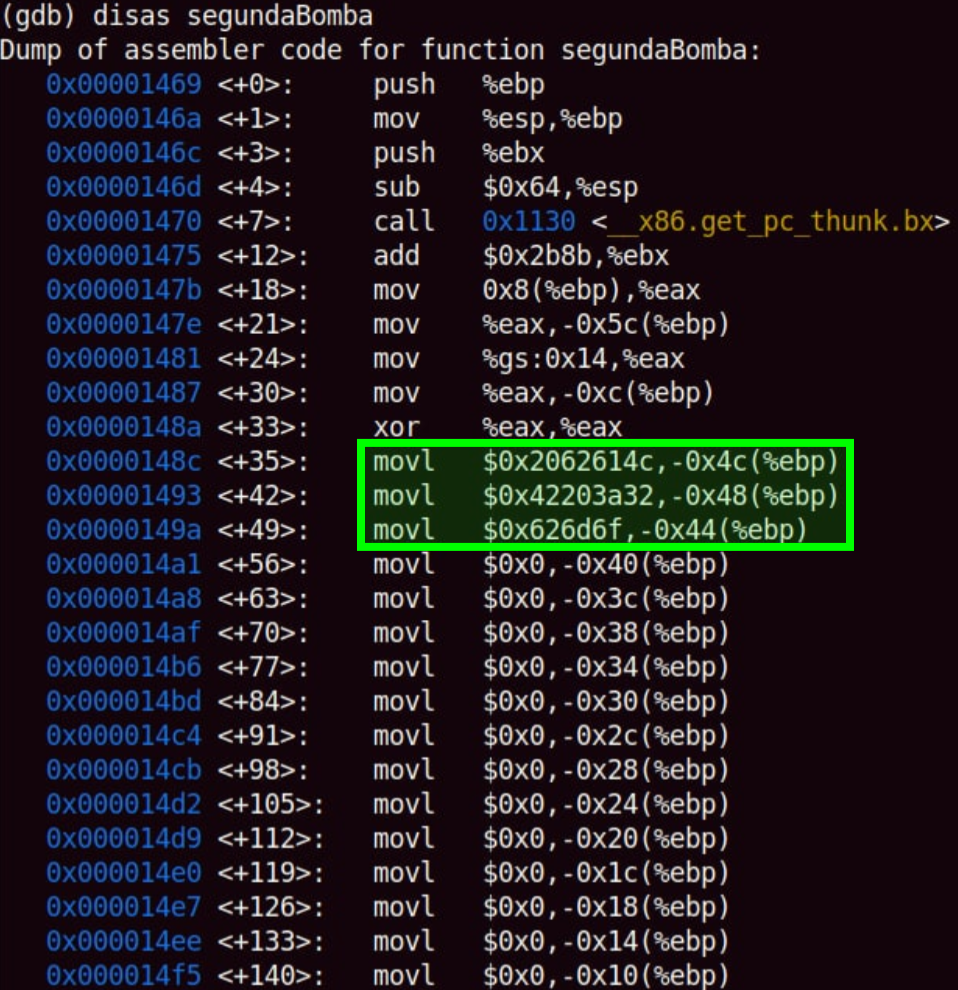
Invertendo cada bloco, chegamos a

Lab 2: Bomb,

que é exatamente a segunda senha.

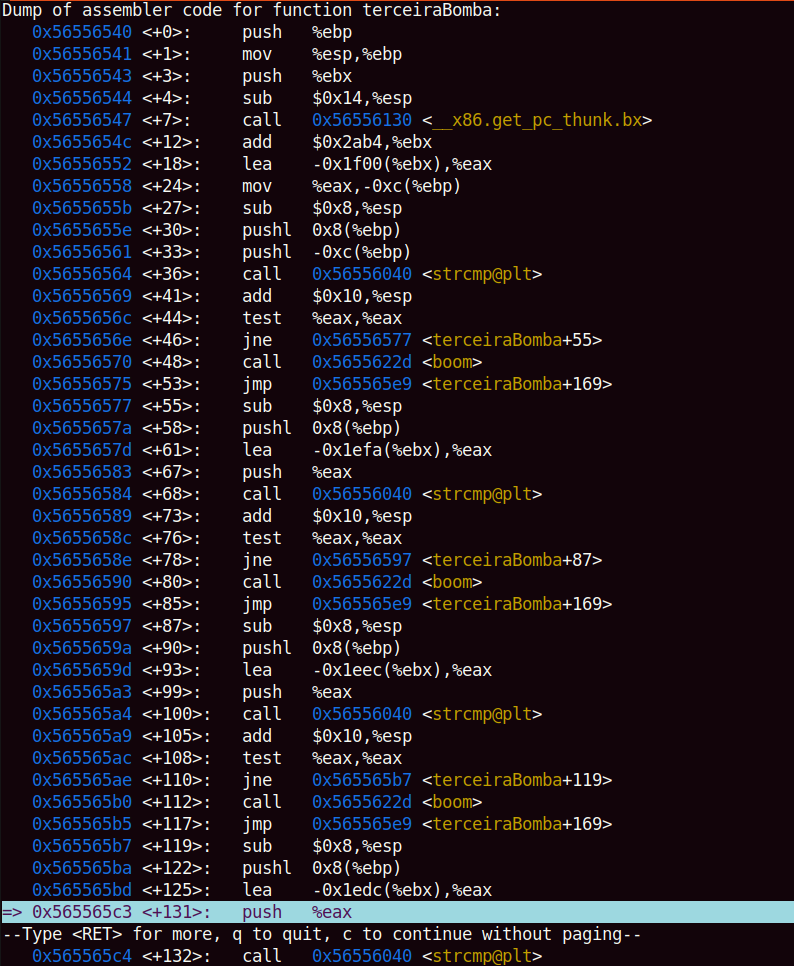
Tivemos de inverter pois:

%eax = (%al + 3) :: (%al + 2) :: (%al + 1) :: (%al + 0)



**Terceira Bomba:**

Nessa bomba notamos que haviam diversas funções strcmp no código. Com isso, nosso raciocínio partiu do princípio de que o a última delas é que faria a comparação se o que digitamos é equivalente à senha esperada. Por isso, utilizamos o comando break \*0x565565C3 (endereço da instrução imediatamente anterior ao último strcmp) para criar um breakpoint nessa instrução. Com isso, imprimimos o conteúdo de %eax para descobrir a senha.



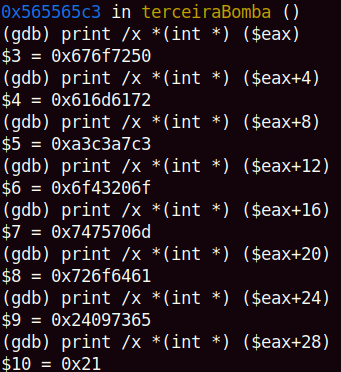
Em destaque na imagem acima está a instrução na qual queremos visualizar o conteúdo. Após isso, digitamos

run 'Mab - 353 - 2020.2' 'Lab 2: Bomb'c

e em seguida

print /x \*(int \*) ($eax)

para de fato visualizar os conteúdos em hexadecimal desejados.



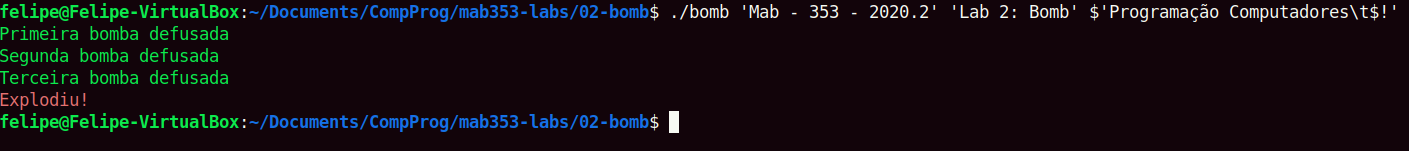
Traduzindo 0x67, 0x6F, 0x72, 0x50, 0x61, 0x6D e assim por diante, até 0x21 para caracteres da tabela ASCII, percebemos que alguns códigos faziam parte da tabela estendida. Então obtemos:

gorP amar oãç [espaço]oC tupm roda $[tab]se !

Paramos em (%eax + 28) pois o conteúdo de (%eax + 32) deixava de fazer sentido. Além disso, pela mesma razão da segunda bomba, ao inverter, chegamos à senha:

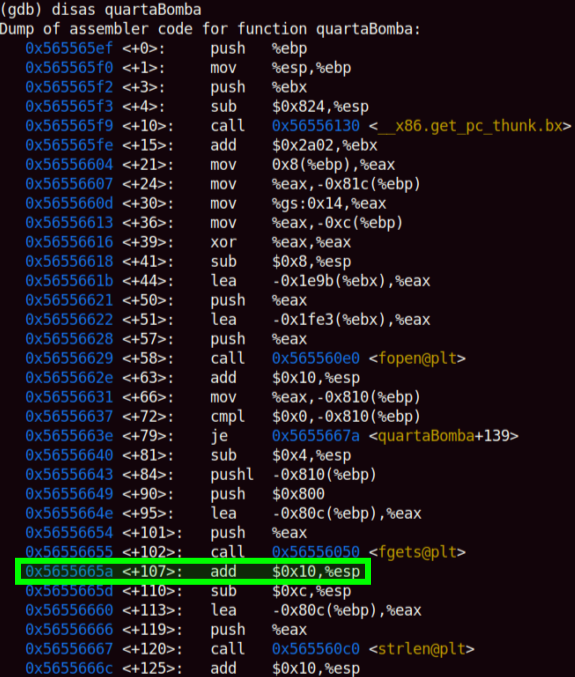
Programação Computadores\t$!

Vale ressaltar que para o computador entender \t como um tab horizontal, devemos colocar o caractere $ antes da senha (que fica entre aspas).



**Quarta Bomba:**

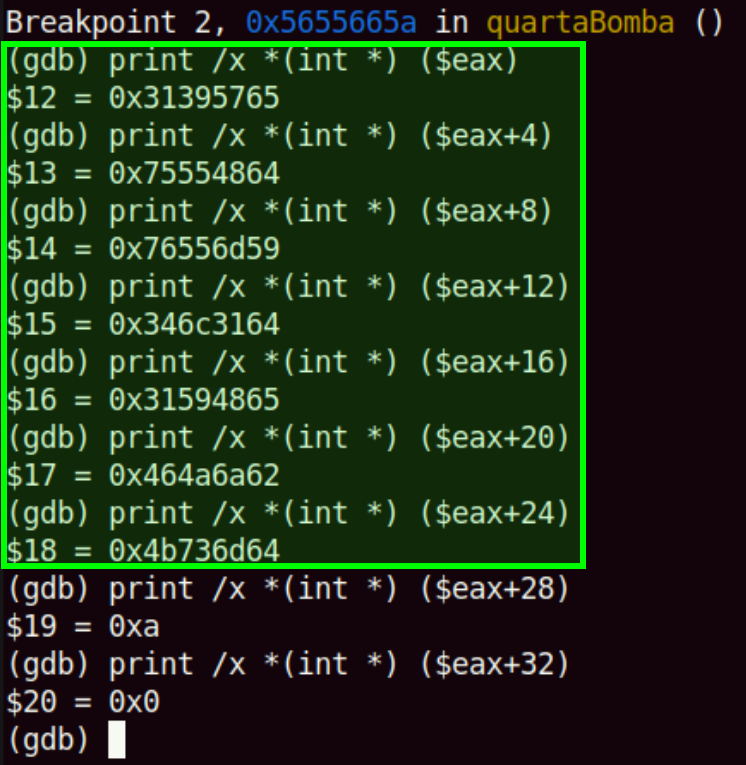
A última bomba nos deixou bastante confusos, pois a senha que descobrimos não era exatamente uma palavra ou frase que fazia sentido. O processo para descobrir a senha partiu do princípio de que o programa estava lendo algo de um arquivo, pois haviam várias funções que lidavam com manipulações de arquivos, tais como fopen, fgets e fclose. A partir dessa constatação, utilizamos o comando break \*0x5655665a para criar um breakpoint imediatamente após a chamada da função fgets na execução do programa (uma vez que ela é responsável por “pegar” algo de um arquivo), para assim, podermos ver os conteúdos de %eax. Fizemos isso pois notamos uma mudança no conteúdo de %eax de antes da chamada para depois dela e, como ele é o registrador acumulador, suspeitamos que a senha poderia estar nele.



Em destaque na imagem acima, está a instrução imediatamente após a função fgets, de onde podemos visualizar os conteúdos desejados. Após isso, utilizamos o comando

run 'Mab - 353 - 2020.2' 'Lab 2: Bomb' $'Programação Computadores\t$!' d

para de fato podermos prosseguir. Depois bastou verificar os conteúdos de %eax até (%eax + 24) para descobrir os números que em ASCII nos dariam a quarta senha.



De cima para baixo, verificamos os correspondentes na tabela para 0x31, 0x 39, 0x57, 0x65, 0x75, 0x55 e assim por diante, até 0x64 (paramos aqui pois em (%eax + 28) já não tinha uma sequência). Fazendo isso, chegamos à seguinte sequência de caracteres:

19We uUHd vUmY 4l1d 1YHe FJjb Ksmd

Novamente, invertendo cada bloco pela razão explicada na segunda bomba, finalmente obtemos a senha

eW91dHUuYmUvd1l4eHY1bjJFdmsK

que decodificada de base64 nos leva para um link de um maravilhoso vídeo do YouTube.

