# PMR3309 - 2016
# 2o Trabalho: Desarmando uma "bomba" binária
# Data de Entrega: 27/09/2018

Adaptado do Material de Bryant e O'Hallaron para o curso 15-213 2016 da CMU

## 1 Introdução

O nefasto *Dr. Evil* armou "bombas binárias" para cada aluno do curso. Uma bomba binária é um programa que consiste em uma sequência de fases. Cada fase espera uma cadeia de caracteres na entrada do programa. Se a cadeia correta é fornecida, a fase é *desarmada* e a bomba passa para a próxima fase. Caso contrário a bomba *explode* escrevendo `"BOOM!!!"` e termina. A bomba é desativada quando cada fase for desarmada.

Cada aluno recebe uma bomba *distinta*. A sua missão é desarmar a sua bomba antes da data limite. Boa sorte!

## Primeiro passo: Obter a sua bomba

Você pode obter a sua bomba apontando um browser para a url:

`http://telesto.mcca.ep.usp.br:15213`

Isso abrirá um formulário de solicitação de bomba a ser preenchido. Digite o seu nome de usuário (ou o seu Número USP), o seu endereço de e-mail e aperte o botão "Submit".

O servidor irá montar a sua bomba e retorná-la ao seu browser em um arquivo `tar` chamado `bombk.tar`, onde $k$ é o número *único* da sua bomba.

ATENÇÃO: A sua bomba é *única*! É *impossível* recuperá-la se for perdida (uma nova solicitação ao servidor iria produzir outra bomba). Guarde uma cópia de sua bomba em um local seguro.

Salve o arquivo em um diretório no qual você pretende fazer o seu trabalho. A seguir descompacte-o com o comando `tar -xvf bombk.tar`. Isso irá criar um diretório chamado `./bombk` com os seguintes arquivos:

- `README`: Identifica a bomba e seu dono.
- `bomb`: A bomba binária executável.
- `bomb.c`: Código fonte da rotina `main` da bomba (inclui saudações do *Dr. Evil*).

ATENÇÂO: Nâo execute a bomba antes de ler TODO este documento! Como você verá, a explosão da bomba pode trazer consequências bastante indesejáveis!

## Segundo passo: Desarmar a bomba

O seu trabalho é desarmar a sua bomba.

A bomba só funciona quando conectada à internet. A bomba possui diversas proteções internas e há rumores de que que mesmo uma desconexão durante a sua execução pode causar uma explosão!

Você pode usar diversas ferramentas para ajudá-lo a desarmar a sua bomba. Veja mais informações na seção **dicas**. Em geral, a melhor maneira de proceder é executar a bomba através de um debugger de modo a examinar e controlar a sua execução.

Cada explosão da sua bomba é notificada ao servidor. Cada explosão causa a perda de 1/2 ponto (até um máximo de 20 pontos) na pontuação final da tarefa. Assim, há consequências ao explodir a sua bomba. Seja cuidadoso... De todo modo, até 4 explosões serão ignoradas na sua nota final.

As primeiras 4 fases valem 15 pontos cada. As fases 5 e 5 são um pouco mais difíceis, e valem 20 pontos cada. Assim a pontuação máxima é de 100 pontos (mas será mesmo?).

Embora as fases sejam progressivamente mais difíceis de se desarmar, a expertise que se ganha passando por uma fase deve compensar este ganho de dificuldade. Note no entanto que a última fase é realmente desafiadora, de modo que você não deve esperar até o último momento para começar.

A bomba ignora linhas de entrada vazias. Se você executar a bomba com um argumento, como por exemplo

```
$ ./bomb psol.txt
```

então ela irá ler linhas de entrada de `psol.txt` até atingir EOF (*End Of File*), e então passará a ler entradas do console. Em um momento de fraqueza, Dr. Evil adicionou esta função para que você não precise re-inserir soluções para as fases que você já desarmou.

## Entrega

ATENÇÃO: Este é um projeto *individual*.

Não há nenhuma entrega explícita. A bomba notificará o servidor automaticamente do seu progresso enquanto você trabalha nela.

Você pode acompanhar o seu progresso no painel da turma na url:

`http://telesto.mcca.ep.usp.br:15213/scoreboard`

Esta página é atualizada continuamente para refletir o progresso em cada bomba. Note que pode haver um pequeno atraso nas atualizações.

## Sugestões *(Por favor leia!)*

Há várias maneiras de se desarmar uma bomba. É possível, por exemplo, examiná-la detalhadamente sem jamais executá-la e descobrir exatamente como ela funciona. Esta é uma técnica evidentemente de grande utilidade (pois não há nenhum risco de explodir a bomba) mas na prática pode ser excessivamente trabalhosa. É possível também rodar a bomba sob um debugger, observar passo a passo o que ela faz e usar esta informação para desarmá-la. Este é possivelmente o modo mais fácil de desarmá-la. Em todo caso, lembre-se que cada explosão para além das 4 "gratuitas" afeta a sua nota. Uma pergunta que o aluno poderia se fazer é: Há alguma maneira de tornar as explosões inconsequentes?

Há diversas ferramentas projetadas para se estudar um executável binário que são úteis neste trabalho. Aqui seguem algumas delas:

- `gdb`

  O debugger GNU. Esta é uma ferramenta de linha de comando disponível em um grande número de plataformas. Ela pode rastrear a execução de um programa linha a linha, examinar conteúdo de memória e registradores, exibir código-fonte e Assembly (lembre-se que apenas o código-fonte da função `main` é fornecido), definir *breakpoints*, *watch points* e escrever scripts.

  A página do livro-texto da disciplina contém um guia de referência rápida para o `gdb` que você pode imprimir: `http://csapp.cs.cmu.edu/2e/docs/gdbnotes-x86-64.pdf`

  Outras sugestões para se usar o `gdb`:

  - Para documentação online, digite "`help`" no prompt de comando do `gdb` ou digite "`man gdb`" ou "`info gdb`" no prompt de comando do Linux. Mais informações sobre um determinado comando podem ser obtidas com "`help`" e o nome do comando.
  - O gdb possui uma interface mista que incorpora na mesma tela um prompt de comando e informações sobre o programa sendo executado, como código-fonte, disasssembly e registradores. Para ativá-la, lance o programa com a opção `-tui`. O *output* do programa pode desfazer a formatação desta interface. O comando `refresh` pode ser usado a qualquer momento para reconstruí-la. As informações apresentadas podem ser alteradas com o comando `layout`.
  - Há alguns programas gráficos que fazem interface com o `gdb`. O mais antigo deles é o `ddd` que na prática não oferece substancialmente mais recursos do que o `gdb` em modo texto. No outro extremo do espectro está a IDE (*Integrated Development Environment*) Eclipse, que oferece uma gama enorme de recursos de visualização da execução do programa. Por outro lado, trata-se de uma ferramenta bastante complexa e o seu aprendizado pode consumir tempo precioso do aluno para este trabalho. Talvez um meio-termo razoável seja rodar o gdb sob o emacs. A interface do emacs com o gdb pode ser ativada com a sequência `<alt>+x gdb`. De todo modo, o exercício foi projetado de forma a ser completável sem nenhuma interface adicional ao gdb!

- `objdump`

  Esta é uma ferramenta que examina executáveis binários estaticamente (ou seja, sem executá-los).

  Alguns modos de operação relevantes para esta atividade são:

  - `objdump -t`

    Este comando exibirá a *tabela de simbolos* do binário. A tabela de símbolos inclui, entre outras informações, o nome das funções internas do executável e seus respectivos endereços. O *Dr. Evil* não é conhecido pela sua sutileza, então talvez informação preciosa possa ser obtida meramente olhando para os nomes das funções!

  - `objdump -d`

    Use este comando para converter para linguagem Assembly o código da bomba. É possível na listagem produzida observar o código associadao a cada função. A simples leitura do código Assembly pode mostrar como uma bomba funciona.

- `strings`

  Esta ferramenta exibe todas as *strings* (Cadeias de caracteres) existentes dentro do binário da bomba.

## Notas sobre convenções de chamadas de função System V AMD64 (ou x86_64)

A especificação System V (seguida por programas compilados para Linux) em um processador com arquitetura AMD64 (também conhecida como x86_64) dita os seguintes comportamentos para uma função:

- Funções que recebem até 6 parâmetros os recebem nos registradores. Os parâmetros são passados na ordem em que se encontram na chamada da função respectivamente nos registradores %rdi, %rsi, %rdx, %rcx, %r8 e %r9. Note que eventualmente uma função usa registradores também para suas variáveis locais, incluindo estes!
- Funções usam o registrador %rax para passar o seu valor de retorno.
- Além de registradores, funções usam também a pilha para armazenar suas variáveis locais. Os endereços de variáveis na pilha são sempre relativos ao registrador %rsp. Por exemplo, a função abaixo:

```
long f(long x, long y)
{
    long v[16];
    long i;
    for(i=0;i<16;i++) v[i] = x*i;
    return v[x&0x0f];
}
```

  É convertida no seguinte código Assembly:

```
f:
      subq    $16, %rsp
      movl    $0, %eax
      jmp     f_test
```

```
  f_loop:
        movq    %rdi, %rdx
        imulq   %rax, %rdx
        movq    %rdx, -120(%rsp,%rax,8)
        addq    $1, %rax
f_test:
        cmpl    $15, %eax
        jle     f_loop
        andl    $15, %edi
        movq    -120(%rsp,%rdi,8), %rax
        addq    $16, %rsp
        ret
```

Note que o registrador %eax é empregado para a variável local `i`. Já a variável local `v` é armazenada na pilha. A posição de `v[i]` é equivalente a `-120(%rsp,%rax,8)`.

- O valor de alguns registradores deve ser preservado por funções, ou restaurado antes do seu retorno. Estes registradores são: %rbx, %rbp, %rsp, %r12, %r13, %r14 e %r15. Todos os outros podem ser modificados livremente pela função. De todo modo, uma função usa a pilha também para salvar o valor destes registradores, de modo a usá-los temporariamente como variáveis locais, ou para salvar o valor de alguns dos registradores restantes *antes* de chamar uma função, caso seja necessário preservar seu conteúdo.

  Por exemplo, a função abaixo:

```
long fatorial(long x)
{
    if(x==0) return 1;
    else return x*fatorial(x-1);
}
```

  É convertida no seguinte código Assembly:

```
  fatorial:
        pushq   %rbx
        movq    %rdi, %rbx
        testq   %rdi, %rdi
        je      fatorial_zero
        leaq    -1(%rdi), %rdi
        call    fatorial
        imulq   %rbx, %rax
        jmp     fatorial_fim
  fatorial_zero:
```

```
        movl    $1, %eax
fatorial_fim:
        popq    %rbx
        ret
```

Note que a função usa o registrador %rbx para armazenar a variável `x`. Como o valor deste registrador deve ser restaurado antes de retornar, a função salva o valor antigo com a instrução `pushq %rbx`. Antes do retorno, o valor original é restaurado pela instrução `popq %rbx`.