GRIS

Grupo de Resposta a Incidentes de Segurança

Olá,

Eu sou Leonardo Ventura

Atualmente membro do GRIS,
desenvolvedor na Equipe SIGA,
bolsista de IC no LabVoIP com a Siemens

Agenda

- Introdução a C
- Introdução a Assembly
- Entendendo a execução de um programa compilado
- O que é engenharia reversa
- Um pouco de engenharia reversa na prática

Introdução a C

- Criada em 1972 por Dennis Ritchie para desenvolvimento do sistema operacional Unix
- Linux é escrito em C
- Vou apresentar o que importa pra gente

Live coding

Introdução Assembly

- Temos um pouco mais de coisa a explicar
- Primeira forma programática de dar instruções a uma máquina
- Em assembly temos apenas instruções pré definidas
- Buscamos conseguir entender um programa em Assembly

Introdução Assembly

- Você programa o processador diretamente
- Ao invés de variáveis, temos registradores!
- Código em assembly não precisa ser compilado, apenas traduzido

Como assim traduzido?

- Em C, precisamos compilar
- Em Assembly, apenas traduzir
- Cada instrução no Assembly é equivalente a uma sequência de números já pré definida

Um programa na memória

- Quando executamos um binário pela shell usando `./programa`, um *loader* é chamado, e o código binário do seu programa é colocado na memória para ser executado
- Uma vez na memória, esse programa irá executar as instruções em Assembly descritas pelos zeros e uns contidos lá dentro

Um programa na memória

- A memória é organizada em **pilha**
- Detalhe: a pilha cresce pra baixo
- Para acessar dados que não estão no topo da pilha, utilizamos seu endereço de memória

Registro de ativação

- Toda função tem o que chamamos de registro de ativação
- Contexto sob o qual ela é executada
- Basicamente, espaço de memória que a função recebe para executar o seu código
- Quando uma função termina, ela precisa deixar o registro de ativação da função que a chamou do mesmo jeito que estava antes de ser chamada

O que são registradores?

- Registradores fazem parte da CPU
- São áreas onde o processador consegue fazer o acesso aos dados de forma mais rápida (topo da hierarquia de memória)
  - Registrador -> Cache -> Memória Principal -> Memória Secundária
- Podemos imaginar como se fossem variáveis de um programa C

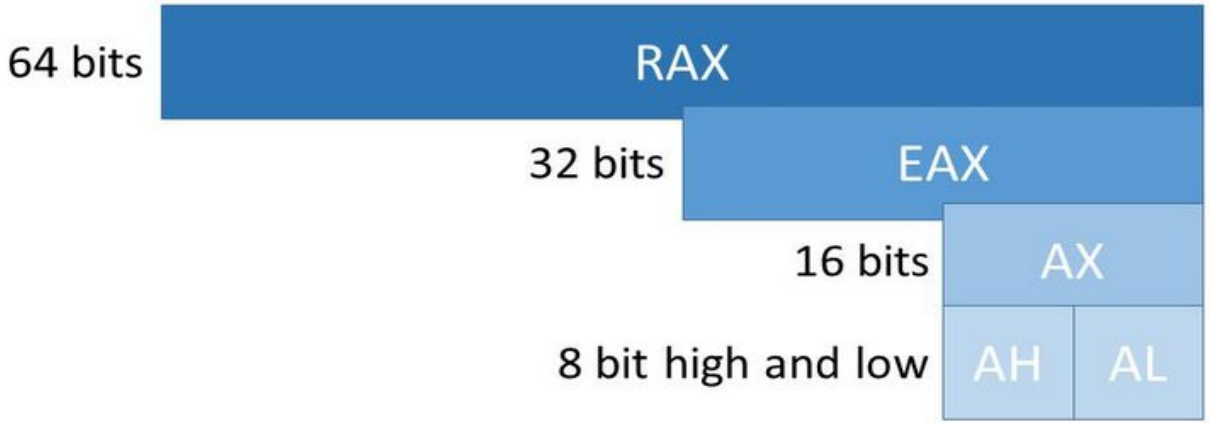
Registradores

- Registradores e nomes variam de arquitetura pra arquitetura (ARM, Intel x86, Intel x64)
- Seguem algum tipo de sentido
- Existem registradores com propósito específico
- Vamos aprender principalmente sobre x86 e x64 (arquiteturas de 32 e 64 bits respectivamente)

Registradores

- Tamanho dos registradores (próximo slide)
- Propósitos:
  - Saber onde está o topo da pilha
  - Saber onde está a base da pilha
  - Saber qual endereço da próxima instrução
  - Saber o retorno de uma função

Registradores



Registradores

- RSP: Stack Pointer, aponta pro topo da pilha
- RBP: Base Pointer, aponta pra base da pilha, início do registro de ativação
- RIP: Instruction Pointer, aponta para o endereço da próxima instrução
- EFLAGS: Registrador de Flags, utilizado para comparações, checagem de erros, etc

Instruções básicas

- Podemos mover um valor pra um registrador
  - `mov rax, 4`
- Podemos somar
  - `add rax, 12`
- Podemos mover pra dentro de um endereço
  - `mov DWORD PTR [rsp], 10`

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x8
mov rsp, rbp
pop rbp
```

	rbp antigo (início do registro de ativação da função que chamou foo)
rsp ->	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x8
mov rsp, rbp
pop rbp
```

	rbp antigo (início do registro de ativação da função que chamou foo)
rsp -> rbp ->	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x8
mov rsp, rbp
pop rbp
```

	0x40	rbp antigo (RA antigo)
rbp ->	0x3C	
rsp ->	0x38	
	0x34	
	0x30	
	0x2C	
	0x28	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x8
mov rsp, rbp
pop rbp
```

	0x40	rbp antigo (RA antigo)
rbp ->	0x3C	
	0x38	12
rsp ->	0x34	
	0x30	
	0x2C	
	0x28	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x8
mov rsp, rbp
pop rbp
```

	0x40	rbp antigo (RA antigo)
rbp ->	0x3C	
	0x38	12
rsp ->	0x34	
	0x30	
	0x2C	
	0x28	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x4
mov rsp, rbp
pop rbp
```

	0x40	rbp antigo (RA antigo)
rbp ->	0x3C	
rsp ->	0x38	12
	0x34	
	0x30	
	0x2C	
	0x28	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x4
mov rsp, rbp
pop rbp
```

	0x40	rbp antigo (RA antigo)
rbp -> **rsp ->**	0x3C 0x3C	
	0x38	12
	0x34	
	0x30	
	0x2C	
	0x28	

Entendendo a pilha

```
foo:
push rbp
mov rbp, rsp
sub rsp, 0x4
push 12
call bar
add rsp, 0x4
mov rsp, rbp
pop rbp
```

	0x40	rbp antigo (RA antigo)
rbp -> **rsp ->**	0x3C 0x3C	
	0x38	12
	0x34	
	0x30	
	0x2C	
	0x28	

Live coding

Parte prática

Prática

- Vocês vão ter o tempo restante do treinamento pra resolver 4 desafios, cada um deles fica gradativamente mais difícil.

Fontes

- https://en.wikipedia.org/wiki/FLAGS_register
- https://stackoverflow.com/questions/29790175/assembly-x86-leave-instruction/29790275

Mais conteúdo

- https://hshrzd.wordpress.com/how-to-start/
- https://malwareunicorn.org/workshops/re101.html#0
- https://www.malwaretech.com/
- https://github.com/rshipp/awesome-malware-analysis
- https://en.wikipedia.org/wiki/FLAGS_register
- https://stackoverflow.com/questions/29790175/assembly-x86-leave-instruction/29790275

OBRIGADO

 leo-ventura@dcc.ufrj.br

 @leoventura98