CET088, II Semestre de 2017
The Attack Lab: Entendendo Bugs de Buffer Overflow
Enviado: Seg, 12 de Dezembro
Entrega: Seg, 14 de Janeiro, 23:59

1 Introdução

Esta atividade envolve a geração de um total de cinco ataques sobre dois programas com diferentes vulnerabilidades de segurança. Os aprendizados que você irá obter neste laboratório incluem:

- Você irá aprender diferentes forma que os atacantes podem explorar as vulnerabilidades de segurança quando programas não se protegem bem contra Buffer Overflows.
- Você terá um melhor entendimento de como escrever programas mais seguros, assim como de algumas características fornecidas pelos compiladores e sistemas operacionais para desenvolver programas menos vulneráveis.
- Você obterá um conhecimento profundo dos mecanismos da pilha (Stack) e da passagem de parâmetros do código de máquina x86-64.
- Você obterá um conhecimento profundo dos mecanismos de como as instruções x86-64 são codificadas.
- Você irá ganhar mais experiência com as ferramentas de depuração, tais como GDB e OBJDUMP.

Note: Neste laboratório, você irá ganhar em primeira mão a experiência com métodos utilizados na exploração das vulnerabilidades de segurança em servidores de rede e sistemas operacionais. A proposta é ajudá-lo a aprender sobre a operação em tempo de execução dos programas e entender a natureza destas vulnerabilidades de tal forma que você possa evitar quando for escrever códigos de sistema. Não será tolerado qualquer outra forma de ataque para obter acesso não autorizado a qualquer recurso do sistema.

As seções 3.10.3 e 3.10.4 do nosso livro texto são materiais de referência para este laboratório.

2 Logistica

Como sempre, este é um projeto individual. Você irá gerar ataques para programas alvos que foram criados especialmente para você.

Ao executar ctarget e rtarget utilizem a opção -q.

Você deverá enviar os arquivos por email, com título: [Attacklab SWB]

Fase 1: primeironome-ctarget.l1, Fase 2: primeironome-ctarget.l2, Fase 3: primeironome-ctarget.l3, Fase 4: primeironome-rtarget.l2, Fase 5: primeironome-rtarget.l3,

Num único pacote em formato .tar

2.1 Obtendo Arquivos

Seus arquivos serão enviados por email

Salve o arquivo `target`k`.tar` num diretório (protegido) Linux no qual você planeja realizar a atividade. Então dê o comando: `tar -xvf target`k`.tar`. Isto irá extrair um diretório `target`k contendo os arquivos descritos abaixo.

Atenção: Se você expandir `target`k`.tar` num computador utilizando um utilitário como Winzip, ou deixar o browser fazer a extração você poderá resetar os bits de permissão dos arquivos executáveis.

Os arquivos em `target`k incluem:

`README.txt`: Um arquivos descrevendo o conteúdo do diretório

`ctarget`: Um programa executável vulnerável ao ataque de *injeção de código*

`rtarget`: Um programa executável vulnerável ao ataque de *programação orientada a retorno*

`cookie.txt`: Um código hexadecimal de 8 digitos que você irá usar como identificador único em seu ataque.

`farm.c`: O código fonte de seu alvo "gadget farm," que você irá utilizar na geração dos ataques de programação orientada a retorno.

`hex2raw`: Um utilitário para gerar strings de ataque.

Nas próximas instruções, assumiremos que você tenha copiado para os arquivos para um diretório local protegido, e que você esteja executando os programas neste diretório local.

2.2 Pontos Importantes

Aqui está um sumário de algumas regras importante sobre a validade das soluções para este laboratório. Estes pontos podem não fazer muito sentido quando você ler este documento na primeira vez. Eles são apresentados aqui como uma referência central das regras assim que você começar.

- Você deve fazer sua tarefa numa máquina que seja similar ao que gerou os seus alvos
- Suas soluções não podem utilizar ataques para contornar o código de validação nos programas. Especificamente, qualquer endereço que você incorpore dentro da string de ataque para usar pela instrução `ret` deverá ser para um dos seguintes destinos:

  - Os endereços para as funções `touch1`, `touch2`, ou `touch3`.
  - O endereço para o seu código de injeção
  - O endereço para um dos seus gadgets vindos do gaget farm (celeiro de gadgets)

- Você poderá somente construitr gadgets do arquivo `rtarget` com faixas de endereços entre as funções `start_farm` e `end_farm`.

3 Programas Alvo

Ambos, CTARGET e RTARGET leêm strings da entrada padrão. Eles fazem isso com a função `getbuf` definida abaixo:

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

A função `Gets` é similar a função da biblioteca padrão `gets`—ela lê uma string da entrada padrão (terminado pelo '`\n`' ou end-of-file) e a armazena (ao longo com o terminador null) no destino específico. Neste código, você pode ver que o destino é uma array `buf`, declarado com o tamanho de `BUFFER_SIZE` bytes. No momento que que seus alvos foram gerados, `BUFFER_SIZE` foi gerado como uma constante em tempo de compilação específica para a sua versão do programa.

A funções `Gets()` e `gets()` não tem como determinar se os seus buffers de destino são grandes o suficiente para armazenar o tamanho da string que elas leeêm. Elas simplesmente copiam as sequências de bytes, possivelmente sobrepondo os limites de armazenamento alocados nos destinos.

Se a string digitada pelo usuário e lida por `getbuf` é pequena o suficiente, `getbuf` irá retornar 1, como mostrado nos próximos exemplos de execução:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Mantendo Curta!
No exploit.  Getbuf returned 0x1
Normal return
```

Um erro ocorrerá se você digitar uma string longa:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Um ninho de mafagafos, com cinco mafagafinhos, ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

(Note que o valor do cookie mostrado irá ser diferente entre cada um de vocês.) O programa RTARGET terá o mesmo comportamento. Como a mensagem indica, a sobreposição do buffer irá causar na maioria das vezes a corrupção do estado do programa, levando a um erro de acesso de memória. Sua tarefa é ser mais inteligente com as strings que você alimenta em CTARGET e RTARGET, de tal forma que eles façam coisas mais interessantes. Estas são chamadas de strings de *exploração*

Ambos CTARGET e RTARGET recebem diferente tipos de argumentos de linha de comando:

`-h`**:** Imprime uma lista dos possíveis argumentos de comando de linha.

`-q`**:** Não envia resultados para o servidor de avaliação.

`-i FILE`**:** Alimenta a entrada de um arquivo, ao invés da entrada. padrão.

Suas strings de exploração serão tipicamente valores contendo bytes que não correspondem aos valores ASCII para impressão de caracteres. O programa HEX2RAW irá ajudá-lo a gerar estas strings brutas (*raw*). Veja o Apêndice A para mais informações de como usar HEX2RAW.

Pontos Importantes:

- Sua string de exploração não poderá conter o valor de byte `0x0a` em qualquer posição intermediária, uma vez que este é o código ASCII para uma nova linha (‘`\n`’). Quando `Gets` encontra este byte, ele irá assumir que você quer terminar a string.
- O programa HEX2RAW espera valores hexadecimais de dois digitos separados por um ou mais espaços em branco. Assim, se você quiser criar um byte com um valor hexadecimal 0, você precisa escrevê-lo como `00`. Para criar a palavra `0xdeadbeef` você deverá passar “`ef be ad de`” para HEX2RAW (note a reversão requerida para a ordem de byte little-endian)

Quando você resolver um dos níveis corretamente, o seu programa alvo irá enviar automaticamente uma notificação para o servidor de avaliação (se habilitado). Por exemplo:

```
unix> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

A Figura 1 apresenta as cinco fases do laboratório. Como você pode ver, as três primeiras fases envolve, o ataque de injeção de código sobre CTARGET, enquanto as duas restantes envolvem o ataque de programação orientada a retorno (ROP) sobre RTARGET.

4 Parte I: Ataque de Injeção de Código

Para as três primeiras fases, suas strings de exploração irão atacar CTARGET. Este programa é configurado de tal forma que as posições da pilha serão consitentes entre uma execução e outra, de tal forma que os dados

Fase	Programa	Nível	Método	Função	Pontos
1	CTARGET	1	CI	`touch1`	10
2	CTARGET	2	CI	`touch2`	25
3	CTARGET	3	CI	`touch3`	25
4	RTARGET	2	ROP	`touch2`	35
5	RTARGET	3	ROP	`touch3`	5

CI: Injeção de Código
ROP: Programação Orientada a Retorno

Figura 1: Sumário das fases do attack lab

na pilha possam ser tratados como um código executável. Esses recursos tornam o programa vulnerável a ataques onde as strings de exploração contêm as codificações de bytes do código executável.

4.1 Nível 1

Para a Fase 1, você não irá injetar um novo código. Ao invés, sua string de exploração irá redirecionar o programa para executar um procedimento existente.

A função `getbuf` é chamada dentro de CTARGET pela função `test` tendo o seguinte código C:

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit.  Getbuf returned 0x%x\n", val);
6 }
```

Quando `getbuf` executa a sua declaração (statement) de retorno (linha 5 de `getbuf`), o programa ordinariamente resume a execução dentro da função `test` (na linha 5 desta função). Queremos mudar este comportamento. Dentro do arquivo `ctarget`, há um código para uma função `touch1` com a seguinte representação em C:

```
1 void touch1()
2 {
3     vlevel = 1;       /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

Sua tarefa é fazer CTARGET executar o código de `touch1` quando `getbuf` executar a sua declaração de retorno, ao invés de retornar para `test`. Note que sua string de exploração pode também corromper partes da pilha de forma não diretamente relatada neste estágio, mas isto não irá causar um problema, desde que `touch1` faça o programa sair diretamente.

Alguns conselhos :

- Toda a informação que você precisa para criar sua string de exploração para este nível podem ser determinadas examinando uma versão desmontada de CTARGET. Utilize `objdump -d` para obter a sua versão desmontada.
- A ideia é posicionar uma representação de byte do endereço de inicio para `touch1` tal que a instrução `ret` ao fim do código para `getbuf` irá transferir o controle para `touch1`.
- Cuidado com a ordem do byte.
- Você pode utilizar o GDB para executar o programa passo a passo através das últimas instruções de `getbuf` para ter certeza que está fazendo a coisa certa.
- A localização de `buf` dentro da moldura de pilha para `getbuf` depende do valor da constante gerada em tempo de compilação `BUFFER_SIZE`, bem como da estratégia de alocação utilizada pelo GCC. Você precisará examinar o código desmontado para determinar sua posição.

4.2 Nível 2

A Fase 2 envolve a injeçã de uma pequena quantidade de código como parte de sua string de exploração.

Dentro do arquivo `ctarget` há o código para a função `touch2` com a seguinte representação em C:

```
1 void touch2(unsigned val)
2 {
3     vlevel = 2;       /* Part of validation protocol */
4     if (val == cookie) {
5         printf("Touch2!: You called touch2(0x%.8x)\n", val);
6         validate(2);
7     } else {
8         printf("Misfire: You called touch2(0x%.8x)\n", val);
9         fail(2);
10    }
11    exit(0);
12 }
```

Sua tarefa é fazer CTARGET executar o código de `touch2` ao invés de retornar para `test`. Neste caso, entretanto, você deve fazer parecer à `touch2` que você tenha passado o seu cookie como argumento.

Alguns conselhos:

- Você deverá posicionar uma representação de byte do endereço do seu código injetado de tal forma que a instrução `ret` no fim do código de `getbuf` irá transferir o controle para ele
- Lembre-se que o primeiro argumento para uma função é passado no registrador `%rdi`.
- Seu código injetado dever configurar o restrador para o seu cookie, e então usar uma instrução `ret` para transferir o controle para a primeira instrução em textttttouch2.

- Não tente utilizar as instruções `jmp` ou `call` em seu código de exploração. As codificações do endereço de destino para estas instruções são dificeis de formular. Utilize as instruções `ret` para todas as transferências de controle, mesmo quando você não está retornando de uma chamada.
- Veja a discussão no Apêndice B sobre como utilizar as ferramentas para gerar as representações em nível de byte das sequências de instruções.

4.3 Nível 3

A Fase 3 envolve um ataque de injeção de código, mas passando uma string como argumento.

Dentro do arquivo `ctarget` há um código para as funções `hexmatch` e `touch3`, possuindo as seguintes representações em C:

```
1 /* Compare string to hex represention of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4     char cbuf[110];
5     /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13     vlevel = 3;       /* Part of validation protocol */
14     if (hexmatch(cookie, sval)) {
15         printf("Touch3!: You called touch3(\"%s\")\n", sval);
16         validate(3);
17     } else {
18         printf("Misfire: You called touch3(\"%s\")\n", sval);
19         fail(3);
20     }
21     exit(0);
22 }
```

Sua tarefa é fazer CTARGET executar o código de `touch3` ao invés de retornar para `test`. Você deve fazer parecer para `touch3` como se você tivesse passado uma representação da string do seu cookie como seu argumento.

Alguns conselhos:

- Você precisará incluir a strind de representação do seu cookie em sua string de exploração. A string deve consistir de oito digitos hexadecimais (ordenado do mais ao menos significante) sem acompanhar o "`0x`."

- Lembre-se que uma string é representada no C como uma sequência de bytes seguida por um byte com valor 0. Digite "`man ascii`" em qualquer máquina Linux para as representações de byte dos caracteres que você precisa.
- Seu código de injeção deve configurar o registrador `%rdi` para o endereço desta string.
- Quando as funções `hexmatch` e `strncmp` são chamadas, elas empilham os dados na Pilha, sobrescrevendo porções da memória que mantiveram usado por `getbuf`. COmo resultado, você irá precisar ser cuidados onde você irá colocar a string de representação de seu cookie.

5 Parte II: Programação Orientada a Retorno

Realizar ataques de injeção de código sobre o programa RTARGET é muito mais dificil do que é para CTARGET, devido a utilização de duas técnicas para frustrar ataques como:

- Utilizar randomização, de tal forma, que a posição da pilha diferem a cada execução. Isto torna impossível determinar onde o seu código injetado será localizado.
- Marcar a seção de memória que armazena a pulha como não executável, assim, mesmo que você consigure o contador do programa para o inicio do seu código injetado, o programa irá falhar com uma uma falta de segmentação.

Felizmente, pessoas inteligentes desevolveram estratégias para se obter algo pronto num programa pela execução de código existente, ao invés de injetar um novo código. A forma mais comum disso é referenciada como programação orientada a retorno *return-oriented programming- ROP* [1, 2]. A estratégia com ROP é identificar sequências de bytes dentro de um programa existente que consistem de uma ou mais instruções seguidas pela instrução `ret`. Um segmento é referenciado como um *gadget*. A Figura 2 ilustra como a pilha pode ser configurada para executar uma sequência de n gadgets. Nesta figura, a pilha contem uma sequência de endereços de gadget. Cada gadget consiste de uma série de bytes de instruções, com o byte final sendo `0xc3`, codificando a instrução `ret`. Quando o programa executa uma instrução `ret` iniciando com esta configuração, ele irá iniciar uma cadeia de execuções de gadget, com a instrução `ret` no fim de cada gadget causando o salto do programa para o inicio do próximo.

Um gadget pode fazer uso do código, correspondendo às declarações em linguagem assembly gerados pelo compiladorm especialmente aqueles que ao fim das funções. Na prática, pode haver alguns gadgets úteis deste formato, mas não o suficiente para implementar muitas operações importantes. Por exemplo, é improvável que uma compilada deva ter em suas últimas instruções `popq %rdi` como sua última instrução antes de `ret`. Felizmente, com um conjunto de instruções orientadas por byte, como x86-64, um gadget pode ser econtrado com a extração de padrões de outras partes da seqüência de bytes de instruções.

Por exemplo, uma versão de `rtarget` contém o código gerado para a seguinte função em C:

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

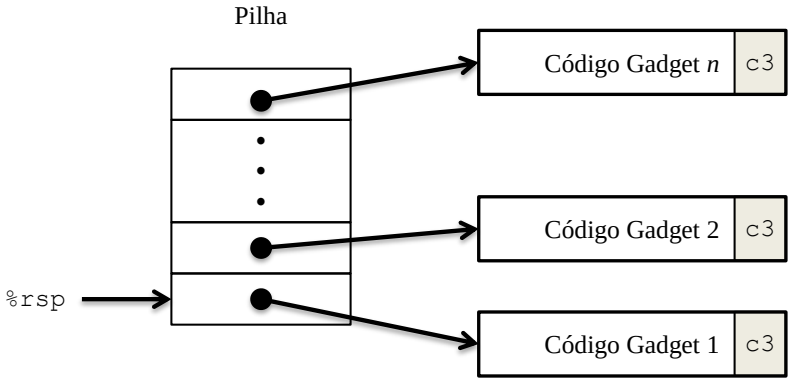

Figura 2: Configurando uma sequência de gadgets para execução. O valor de byte `0xc3` codifica a instrução `ret`

As chances desta função ser útil para atacar um sistema parace ser pequena. Mas, o código de máquina desmontado para esta função mostra uma sequência de bytes interessante:

```
0000000000400f15 <setval_210>:
  400f15:       c7 07 d4 48 89 c7       movl   $0xc78948d4,(%rdi)
  400f1b:       c3                      retq
```

A sequência `48 89 c7` codifica a instrução `movq %rax, %rdi`. (Veja a Figura 3A para codificação de instruções `movq` úteis.) Esta sequência seguida pelo byte `c3` que codifica a instrução `ret`. A função começa no endereço `0x400f15`, e a sequência começa no quarto byte da função. Assim, este código contém um gadgetm tendo um endereço de inicio `0x400f18`, que irá copiar o valor de 64-bits no registrador `%rax` para o registrador `%rdi`.

Seu código para RTARGET contém um número de funções similares para a função `setval_210` mostrada acima numa região definida como *celeiro de gadget (gadget farm)*. Seu trabalho será identificar gadgets úteis no celeiro de gadget e utilizá-los para realizar ataques similares aos que você fez nas fases 2 e 3.

Importante: O celeiro de gadgets é demarcado pelas funções `start_farm` e `end_farm` em sua cópia de `rtarget`. Não tente construir gadget de outras partes do código de programa.

5.1 Nível 2

Para a Fase 4, você irá repetir o ataque da Fase 2, mas apenas no programa RTARGET utilizando gadgets de seu celeiro de gadget. Você pode construir sua solução utilizando gadgets que consistem dos seguintes tipos de instruções e utilizar apenas os oito primeiros registradores x86-64 (`%rax`–`%rdi`).

`movq` : O seu código é mostrado na Figura 3A.

`popq` : O seu código é mostrado na Figura 3B.

`ret` : Esta instrução é codificada pelo único byte `0xc3`.

`nop` : Esta instrução (pronunciada "no op," acrônimo de "no operation") é codificada pelo único byte `0x90`. Seu efeito é apenas causar o incremento de 1 no contador do programa.

Alguns conselhos:

- Todos os gadgets que você precisa podem ser encontrados na região do código para `rtarget` demarcado pelas funções `start_farm` e `mid_farm`.
- Você pode realizar este ataque com apenas dois gadgets.
- Quando um gadget utiliza uma instrução `popq`, ele irá desempilhar dados da pilha. Como resultado, sua string de exploração irá conter uma combinação de endereços de gadget e dados.

A. Codificação das instruções `movq`

`movq` S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Codificação das instruções `popq`

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

C. Codificação das instruções `movl`

`movl` S, D

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Codificação das instruções nop de 2 bytes

Operation	Register R			
	%al	%cl	%dl	%bl
andb R, R	20 c0	20 c9	20 d2	20 db
orb R, R	08 c0	08 c9	08 d2	08 db
cmpb R, R	38 c0	38 c9	38 d2	38 db
testb R, R	84 c0	84 c9	84 d2	84 db

Figura 3: Codificação em Byte das instruções. Todos os valores são mostrados em hexadecimal.

5.2 Nível 3

Antes de começar a Fase 5, pare para considerar o que você fez até aqui. Nas Fases 2 e 3, você fez um programa executar um código de máquina feito por você. Se CTARGET tivesse sido feito num servidor de rede, você poderia ter injetado seu código numa máquina remota. Na Fase 4, você contornou dois dos principais dispositivos utilizados nos sistemas modernos para evitar ataques de Buffer Overflow. Embora você não tenha injetado o seu próprio código, você conseguiu injetar um tipo de programa que funciona ao combinar sequências de código existente. Você também conseguiu 95/100 pontos para o laboratório. Esta é uma boa pontuação. Se você possui outras obrigações sobre pressão considere parar por aqui.

A Fase 5 requer que você faça um ataque ROP sobre RTARGET para invocar a função `touch3` com um ponteiro para uma representação de string de seu cookie. Isso pode não ser significantemente mais dificil que utilizar um ataque ROP para invocar `touch2`, exceto o fato de ter sido feito assim. Além disso, a Fase 5 dá apenas 5 pontos, o que não é uma medida de dificuldade para o esforço que irá requerer. Pense nisso como um problema de crédito extra para aqueles que querem ir além das expectativas normais para este curso.

Para resolver a Fase 5 você deverá utilizar gadgets ba região do código em `rtarget` demarcado pelas funções `start_farm` e `end_farm`. Em adição aos gadgets utilizados na Fase 4, este celeiro expandido inclue as codificações de diferentes instruções `movl`, como mostrado na Figura 3C. As sequências de byte nesta parte do celeiro também contém instruções de 2-bytes que servem como *nops funcionais*, i.e., eles não realizam qualquer mudança de valor em registrador ou memória. Estes incluem instruções, mostrados na Figura 3D, como `andb %al,%al`, que opera em bytes de baixa ordem de alguns dos registradores mas não mudam os seus valores.

Alguns conselhos:

- Você poderá revisar o efeito que uma instrução `movl` tem sobre os 4 bytes mais significativos de um registrador.
- A solução oficial requer oito gadgets (nem todas são únicos)

Boa Sorte e divirta-se!

A Utilizando HEX2RAW

HEX2RAW pega como uma entrada uma string *hex-formatada*. Neste formato cada valor de byte é representado por dois digitos hexadecimais. Por exemplo, a string "`012345`" deve ser digitada como no formato hex "`30 31 32 33 34 35 00`." (Lembre que o código ASCCI para digitos decimais x é `0x3`x, e que o fim da string é indicado por um byte null.)

Os caractre hex que você passa para HEX2RAW deverão ser seprados por um espaço em branco. Recomenda-se separar diferentes partes de sua string de exploração com novas linhas enquanto você estiver trabalhando sobre ela. HEX2RAW suporta comentérios em bloco no estilo C, então você pode desabilitar seções de sua string de exploração. Por exemplo:

```
48 c7 c1 f0 11 40 00 /* mov    $0x40011f0,%rcx */
```

Tenha certeza de deixar espaço envolta de ambos, inicio e fim, das strings de comentário (“/*”, “*/”), assim os comentários serão ignorados.

Se você gerar uma string de exploração formatada em Hex no arquivo `exploit.txt`, você pode aplicar a string pura em CTARGET ou RTARGET em diferentes formas:

1. Você pode configurar uma série de pipes para passar a string através de HEX2RAW.

   ```
   unix> cat exploit.txt | ./hex2raw | ./ctarget
   ```

2. Você pode armazenar uma string num arquivo e utilizar o redirecionador de I/O:

   ```
   unix> ./hex2raw < exploit.txt > exploit-raw.txt
   unix> ./ctarget < exploit-raw.txt
   ```

   Este formato pode ser utilizado quando estiver executando dentro do GDB:

   ```
   unix> gdb ctarget
   (gdb) run < exploit-raw.txt
   ```

3. Você pode armazenar a string pura num arquivo e usar o norme do arquivo como um argumento de linha de comando:

   ```
   unix> ./hex2raw < exploit.txt > exploit-raw.txt
   unix> ./ctarget -i exploit-raw.txt
   ```

   Esta forma também pode ser utilizada quando estiver dentro do GDB.

B Gerando Byte Codes

Usando GCC como um montador e OBJDUMP como uma desmontador torna conveniente a geração dos byte codes para as sequências de instruções. Por exemplo, suponha que você escreva um arquivo as an assembler and OBJDUMP as a disassembler `example.s` contendo o seguinte código assembly:

```
# Example of hand-generated assembly code
        pushq   $0xabcdef          # Push value onto stack
        addq    $17,%rax           # Add 17 to %rax
        movl    %eax,%edx          # Copy lower 32 bits to %edx
```

O código pode conter uma mistura de instruções e dados. Qualquer coisa à direita de um caracter ‘#’ é um comentário.

Você pode agora montar e desmontar este arquivo:

```
unix> gcc -c example.s
unix> objdump -d example.o > example.d
```

O arquivo gerado `example.d` contém o seguinte:

```
example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 68 ef cd ab 00        pushq  $0xabcdef
   5: 48 83 c0 11           add    $0x11,%rax
   9: 89 c2                 mov    %eax,%edx
```

As linhas abaixo mostram o código de máquina gerado a partir de instruções da linguagem assembly. Cada linha possui um número hexadecimal à esquerda, indicando o endereço de inicio da instrução (iniciando com 0), enquanto os digitos hex após o caractere '`:`' indica os byte codes para a instrução. Assim, podemos ver que a instrução `push $0xABCDEF` possui um byte code formatado em hex de `68 ef cd ab 00`.

A partir deste arquivo, você pode pegar a sequência de byte para o código:

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

Esta string pode então ser passada através de HEX2RAW para gerar uma string de entrada para os programas alvos. Alternativamente, você pode editar o example.d para omitir valores estranhos e conter comentários no estilo C para melhor legibilidade, resultando em:

```
   68 ef cd ab 00   /* pushq  $0xabcdef  */
   48 83 c0 11      /* add    $0x11,%rax */
   89 c2            /* mov    %eax,%edx  */
```

Isto é também uma entrada válida que você pode passar através de HEX2RAW antes de enviar a um de seus programas alvo.

Referências

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15(1):2:1–2:34, March 2012.

[2] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.