PMR3309 - 2022
O *Attack Lab*: Compreendendo vulnerabilidades causadas por bugs de *buffer overflow*
Data final de entrega: 09/11/2023

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

1 Introdução

Este trabalho envolve a geração de cinco ataques em dois programas distintos que possuem vulnerabilidades de segurança Os objetivos didáticos são:

- Aprender como atacantes podem explorar vulnerabilidades quando programas contém bugs de *buffer overflow* e o (extenso) alcance dos ataques.
- Ganhar um melhor conhecimento de como escrever programas mais seguros, bem como compreender recursos de segurança oferecidos por compiladores e sistemas operacionais (e suas limitações).
- Aprofundar o conhecimento da pilha e mecanismos de passagem de parâmetros de processadores x86-64.
- Compreender a codificação de instruções x86-64.
- Ganhar mais experiência com ferramentas como GDB.

Nota: Nesta atividade você ganhará experiência em primeira mão em métodos reais para explorar vulnerabilidades em sistemas operacionais modernos. O objetivo dela é ajudá-lo a aprender sobre a execução de programas e das vulnerabilidades destes de modo a evitá-las quando escrever seu código. *A coordenação de PMR3309 não aprova o uso destes métodos ou de quaisquer outros para obter acesso indevido a sistemas computacionais.*

As seções 3.10.3 e 3.10.4 do livro-texto são a referência para esta atividade.

1.1 Obtendo os arquivos

Você pode obter os arquivos desta atividade na url:

`http://200.144.254.191:15513/`

O servidor irá gerar os seus arquivos e retorná-los em um pacote `tar` chamado `target`k`.tar`, onde k é o número único dos seus programas-alvo.

Nota: Armazene com cuidado o seu arquivo! Como no bomblab, ele não pode ser re-gerado!

Salve o arquivo `target`k`.tar` em um local protegido. A seguir, descompacte-o com o comando `tar -xvf target`k`.tar`. Isso irá extrair em um diretório `target`k contendo os arquivos descritos adiante.

Atenção: Caso use outra ferramenta para descompactar os arquivos, verifique as permissões de execução nos arquivos binários..

O diretório `target`k contém:

`README.txt`: Um arquivo que descreve o conteúdo do diretório.

`ctarget`: Um executável vulnerável a ataques do tipo *code-injection*.

`rtarget`: Um executável protegido com randomização de endereço e pilha não-executável (que deve ser atacadp via *return-oriented programming*).

`cookie.txt`: Um número hexadecimal de 8 dígitos que você usará como um identificador único para seus ataques.

`farm.c`: O código fonte da "*gadget farm*", o trecho de código a ser usado nos ataques de *return-oriented programming*.

`hex2raw`: Uma ferramenta para produzir entradas de ataques.

1.2 Aspectos Importantes

Aqui está um sumário das regras importantes para uma solução ser considerada *válida*. Talvez algumas destas regras não façam sentido em uma primeira leitura. Volte a elas quando estiver construindo a sua solução..

- Você deve fazer o trabalho em uma máquina similar à usada para gerar os seus programa-alvo (uma máquina linux com cpu x86_64).
- Seus ataques não devem contornar o código de validação presente em seus programas. Mais especificamente, qualquer código executado como parte de seu ataque deve estar em algum destes lugares:
  - O endereço *de entrada* das funções `touch1`, `touch2`, ou `touch3`.
  - O seu código injetado na pilha.
  - Código binário presente na *gadget farm*.
- Você só pode construir *gadgets* do binário `rtarget` com endereços entre os símbolos `start_farm` e `end_farm`.
- Algumas fases adicionam restrições específicas.

2 Sobre ataques de buffer overflow

Um ataque de buffer overflow tipicamente subverte o funcionamento de um programa de forma a fazer o que o atacante deseja. A depender da natureza do programa atacado, é possível ler dados da memória, modificá-los, ler e escrever arquivos no disco, abrir novas portas de rede e outros. Tipicamente, a função de um ataque destes é abrir uma "primeira porta" no sistema a partir da qual o atacante pode injetar novos programas maliciosos. Há hoje diversos computadores na internet que foram subvertidos e controlados por atacantes com recursos como estes (e outros mais). Estes computadores, cuja invasão é ignorada pelos seus operadores, fazem parte das chamadas "botnets", vastas redes de computadores controladas por agentes maliciosos que as usam com diversos propósitos (frequentemente, plataformas para novos ataques).

A invasão neste experimento se limitará a executar funções específicas dos programas atacados de forma não-prevista pelo código original, mas as técnicas aqui empregadas são essencialmente as mesmas empregadas na subversão e tomada de controle hostil de máquinas remotas.

3 Programas-Alvo

Ambos CTARGET e RTARGET lêem entradas do *standard input* (tipicamente o console caso não haja nenhuma redireção). Eles o fazem com a função `getbuf` definida a seguir:

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

A função `Gets` é similar à função C `gets`—ela lê uma string do *standard input* (encerrada pelo caractere de nova linha '`\n`' ou fim de arquivo (EOF)) e armazena-a (acrescida de um caractere nulo ao seu final) no destino especificado. Neste código você pode ver que o destino é um vetor `buf`, declarado como tendo `BUFFER_SIZE` bytes. No momento de geração dos seus programamas, `BUFFER_SIZE` foi gerado como uma constante em tempo de compilação.

As Funções `Gets()` e `gets()` não têm como determinar se seus buffers são grandes o suficiente para armazenar as strings que elas lêem. Elas simplesmente copiam sequências de bytes, possivelmente sobrepondo os limites do armazenamento alocado no destino.

Se a string fornecida pelo usuário e lida por `getbuf` for suficientemente pequena, claramente `getbuf` retornará 1, como visto nestes exemplos:

```
user@pmr3309:~/target1$ ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit.  Getbuf returned 0x1
Normal return
```

Tipicamente um erro ocorre se for fornecida uma string longa o suficiente:

```
user@pmr3309:~/target1$ ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

O programa RTARGET se comporta da mesma maneira. Como a mensagem de erro indica, sobrescrever o buffer tipicamente corrompe o estado do programa, o que leva a um erro de acesso de memória e ao término da execução do programa. Sua tarefa é ser mais abilidoso com as strings que você fornece a CTARGET e RTARGET de modo a produzir efeitos mais interessantes. Estas são chamadas de *exploit strings*.

Ambos CTARGET e RTARGET podem tomar diferentes argumentos na sua execução:

`-h`**:** Exibe a lista de possíveis comandos.

`-q`**:** Não envia resultados ao servidor.

`-i FILE`**:** Lê a entrada de um arquivo ao invés do standard input.

Suas stringss de exploit tipicamente conterão bytes que não correspondem a caracteres que possam ser inseridos pelo teclado. O programa HEX2RAW permitírá gerar strings com tais bytes.. Vide apêndice A para mais informação sobre como usar HEX2RAW.

Notas:

- Seu exploit não deve conter o byte `0x0a` em nenhuma posição intermediária, visto que este é o código ASCII para nova linha (‘`\n`’). Quando `Gets` encontra este byte, presume que o usuário quer encerrar a string.
- HEX2RAW espera valores hexadecimais de dois dígitos separados por um ou mais espaços em branco. Assim, se você quer criar um byte com o valor hexadecimal de 0, você deve escrevê-lo como `00`. Para criar a palavra de 32 bits `0xbebacafe` você deve passar “`be ba ca fe`” para HEX2RAW (note a reversão necessária em processadores little-endian).

Quando você resolver corretamente um nível, o programa-alvo irá automaticamente enviar uma notificação ao servidor. Por exemplo:

```
user@pmr3309:~/target1$ ./hex2raw < exploit.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!
```

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

CI: Code injection
ROP: Return-oriented programming

Figura 1: Sumário de fases do attack lab

O servidor irá testar o seu exploit[1] para verificar se ele realmente funciona e irá atualizar a página de score do Attacklab para indicar que o seu alvo (identificado pelo número) foi atacado com sucesso.

Você pode ver o score na página

```
http://200.144.254.191:15513/scoreboard
```

Nota: Ao contrário do bomb lab, *não há penalidade* para erros nesta atividade. Sinta-se a vontade para disparar em CTARGET e RTARGET quaisquer strings que você queira (de fato, a menos que o alvo ache sua string "interessante", ele sequer notificará o servidor).

A figura 1 sumariza as 5 fases da atividade. Como se pode ver, as 3 primeiras envolves ataques de Injeção de Código (IC) no programa CTARGET, enquanto que as duas últimas envolvem ataques de Return-Oriented Programming (ROP) em RTARGET.

4 Parte 1: Ataques de Injeção de Código

Para as três primeiras fases, suas exploit strings irão atacar CTARGET. Este programa foi construído de forma que a pilha estará em posições fixas e dados na pilha são marcados como executáveis. Tais características tornam-no particularmente vulnerável a ataques de injeção de código. Programas mais recentes não possuem estas características, mas há ainda muito código em execução hoje construído desta forma!

Atenção: Usando o ctarget em kernels com randomização de endereços

Versões mais recentes do kernel linux podem carregar o seu programa em *endereços aleatórios* na memória. Isso é feito precisamente para proteger os programas do tipo de ataque que estamos tentando construir! Explicar como contornar esse tipo de proteção vai além do escopo desta atividade. Se observar que seu exploit funciona dentro do debugger, por exemplo, mas não funciona fora dele, a randomização de endereços pode ser o problema. Para rodar um programa sem randomização, use o comando:

[1] Sim, de verdade! Eu passei uma semana inteira tentando construir um ambiente seguro para estes testes. Aparentemente ninguém na CMU deu bola para isso...
– Thiago

```
setarch `uname -m` -R programa-a-ser-executado
```

4.1 Fase 1 (ou `ctarget` – *Level 1*)

Para a Fase 1, você na prática não injetará novo código executável. Ao invés disso, sua exploit string irá redirecionar o programa para executar uma função pré-existente.

A função `getbuf` é chamada dentro de CTARGET por uma função `test` que tem 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 instrução `return` (linha 5 de `getbuf`), o programa tipicamente retoma a execução dentro da função `test` (na linha 5). O objetivo é modificar este comportamento. Dentro do binário `ctarget`, há uma função `touch1` que tem o seguinte código 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 }
```

Em condições normais de funcionamento de CTARGET, a função `touch1` *nunca* é executada. Sua tarefa é forçar CTARGET a executar o código `touch1` quando `getbuf` executa a sua instrução de retorno, ao invés de retornar a `test`. Note que o seu exploit pode também corromper partes da pilha não diretamente relacionadas a esta tarefa, mas isso não causará problemas visto que `touch1` faz com que o programa encerre imediatamente.

Sugestões:

- Boa parte das informações que você precisa para projetar seu exploit podem ser obitdas examinando-se uma versão desmontada de CTARGET. Use `objdump -d` para obter uma listagem em linguagem de montagem de CTARGET.
- A idéia é posicionar uma representação em bytes do endereço de entrada de `touch1` de forma que a instrução `ret` ao final de `getbuf` transfira a execução para `touch1`.
- Atenção para a ordem dos bytes.
- Em algumas situações o endereço na memória das funções uma vez que o programa é carregado pode ser distinto do visto em `objdump -d`. Use o GDB para verificar os endereços do programa durante

a sua execução. Você talvez também queira usar o GDB para verificar os passos do programa nas últimas instruções de `getbuf` para certificar-se de que ele está fazendo o que você deseja.

- O posicionamento da variável `buf` dentro da pilha para `getbuf` depende do valor da constante `BUFFER_SIZE`, bem como da estratégia de alocação empregada pelo GCC. Você deve examinar o código desmontado para determinar a posição adequada.

4.2 Fase 2 (ou `ctarget` – *Level 2*)

A fase 2 envolve a injeção de um pequeno trecho de código na sua exploit string.

Dentro do arquivo `ctarget` há código para uma função `touch2` que tem 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 forçar CTARGET a executar o código para `touch2` ao invés de retornar a `test`. Neste caso, no entanto, você deve invocar `touch2` passando a ela um argumento, o seu "cookie".

Sugestões:

- Novamente, você deve posicionar uma representação em bytes do endereço de entrada de `touch1` de forma que a instrução `ret` ao final de `getbuf` desvie a execução do código. Note que este desvio pode ser feito para um trecho da própria string injetada..
- Lembre-se que o primeiro argumento para uma função é passado no registrador `%rdi`.
- Seu código injetado tem duas tarefas: Definir o argumento de chamada para seu cookie e invocar a função `touch2`.
- Embora pareçam tentadoras, as instruções `jmp` e `call` na prática são difíceis de serem usadas em um exploit. Estas funções usam endereçamento *relativo* ao contador de instruções de forma que a codificação de um desvio para uma posição precisa no código é trabalhosa. Uma idéia melhor em um exploit é usar instruções `ret` para todos os desvios de controle, mesmo quando você não está efetivamente retornado de uma chamada de função (lembre-se que as instruções `ret` recuperam endereços absolutos diretamente da pilha).
- Veja a discussão no apêndice B com sugestões de como gerar representações em bytes de sequências de instruções.

4.3 Fase 3 ou `ctarget` – *Level 3*)

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

Dentro do arquivo `ctarget` há código para as funções `hexmatch` e `touch3` que têm 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 forçar CTARGET a executar o código em `touch3` ao invés de retornar a `test`. Você deve invocar `touch3` passando a ela como argumento uma representação em string do seu cookie.

Sugestões:

- Naturalmente, seu exploit deve incluir uma string com uma representação hexadecimal do seu cookie. A string deve incluir oito dígitos hexadecimais (ordenados do mais significativo para o menos) *sem* o prefixo "`0x`."
- Lembre-se que uma string em C é uma sequência de bytes seguida de um byte com o valor zero. Digite "`man ascii`" em uma máquina Linux para ver a representação em bytes dos caracteres que você precisa.
- Você deve passar como parâmetro para a função o endereço desta string.
- Lembre-se que quando as funções `hexmatch` e `strncmp` são invocadas, elas inserem novos dados na pilha, sobrescrevendo inclusive porções da memória do buffer usado por `getbuf`. Considere cuidadosamente onde você colocará a string no seu exploit!

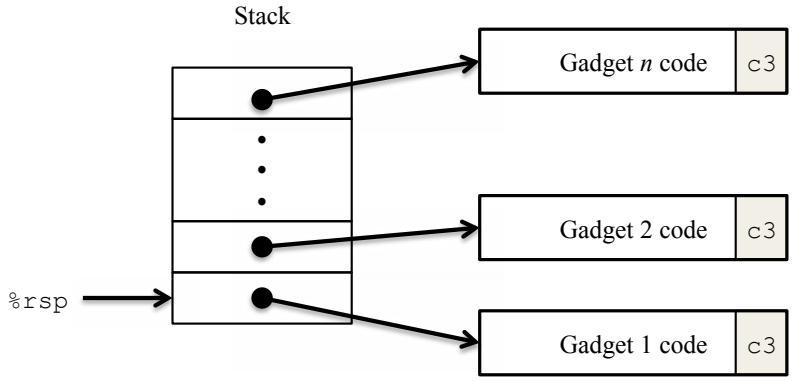

Figura 2: Preparando uma sequência de *gadgets* para execução. O valor `0xc3` codifica a instrução `ret`.

5 Parte II: Ataques baseados em *Return-Oriented Programming*

O programa RTARGET usa técnicas contemporâneas de proteção contra ataques de injeção de código. Estas técnicas são:

- Randomização de endereços: Os endereços da pilha são diferentes e imprevisíveis a cada execução. Isso faz com que seja impossível determinar onde o código injetado estará localizado.
- Pilha Não-Executável: A seção de memória que contém a pilha é marcada pelo sistema operacional como não-executável. Assim, mesmo que você consiga dirigir o contador de programa para a posição adequada no seu código injetado, o programa simplesmente seria encerrado com uma *segmentation fault*.

Felizmente(?), programadores habilidosos criaram estratégias para forçar programas assim protegidos a realizarem tarefas não-previstas sem injetar diretamente código novo. A forma mais geral destas estratégias é conhecida como *Return-Oriented Programming* (ROP) [1, 2]. A estratégia ROP é identificar no código binário existente sequências de bytes que codificam instruções interessantes seguidas de uma instrução `ret`.

Uma tal sequência é chamada de *gadget*[2]. A Figura 2 ilustra como a pilha pode ser preparada para executar uma sequência de n gadgets. Nesta figura, a pilha contém uma sequência de endereços de gadgets.

Cada gadget consiste em uma série de bytes que codificam sequências de instruções encerradas por um `ret`, codificado pelo byte `0xc3`. Quando um programa com uma pilha assim preparada executa um `ret`, ele inicia uma cadeia de execução de gadgets de modo que cada instrução `ret`faz com que a execção pule para o próximo gadget. Um gadget pode usar instruções pré-geradas por um compilador, especialmente aquelas ao final de funções (ou até mesmo funções inteiras!). Na prática, embora possam existir gadgets úteis nesta forma, estes podem revelar-se insuficientes para todas as operações desejadas. Por exemplo, é improvável que uma função escrita em C, uma vez compilada, tenha a instrucão `popq %rdi` como sua

[2] "dispositivo", mas essa tradução não é usual —Thiago

penúltima instrução (antes de `ret`). Por outro lado, em um conjunto de instruções como o x86-64, um gadget pode ser obtido extraindo-se codificações alternativas de trechos de codificações de outas instruções.

Por exemplo, uma versão de `rtarget` poderia conter código gerado pela compilação da seguinte função:

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

Este código não parece de grande serventia para atacar um sistema. No entanto, sua compilação para a arquitetura x86-64 revela 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` que antecede a instrução `ret` codifica a instrução `movq %rax, %rdi`. (Vide a Figura 3A para codificações de instruções movq.) Esta funçãao começa no endereço `0x400f15` e a sequência de interesse começa no 4o byte da função. Deste modo, este código contém um gadget potencialmente útil no endereço `0x400f18` que copia o valor de 64-bits do registrador %rax para o registrador %rdi. Seu código para RTARGET contém diversas funções similares à função `setval_210` em uma região chamada *gadget farm*. Sua tarefa será a de identificar gadgets úteis na gadget farm e usá-los para fazer ataques similares aos das fases 2 e 3. *Importante*: A gadget farm está demarcada pelos símbolos `start_farm`e `end_farm` no seu programa rtarget. Não tente construir gadgets de outras porções do código do programa!

5.1 Fase 4 (Ou `rtarget` - *level 2*)

Para a Fase 4 você irá repetir o ataque da Fase 2, mas o fará no programa RTARGET empregando gadgets da sua gadget farm. Além do código das funções completas (que não parecem ser exatamente relevantes) você pode construir sua solução usando gadgets compostos pelas seguintes instruções, e usando apenas os 8 primeiros registradores (`%rax`–`%rdi`).[3].

`movq` : A codificação para estas está na Figura 3A.

`popq` : A codificação para estas está na Figura 3B.

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

`nop` : Esta instrução (“no op,”, abreviação de “no operation”) é codificada pelo byte `0x90`. O seu único efeito é fazer o contador de programa avançar de 1.

Sugestões:

[3]Dica: Estas restrições limitam a sua solução, mas ao fazê-lo, elas facilitam a sua tarefa! – Thiago

A. Codificações de 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. Codificações de instruções `popq`

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

C. Codificações de 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. Codificações de “nop funcionais” 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: Codificações em bytes de instruções. Todos os valores estão em hexadecimal.

- Você pode construir um exploit bem-sucedido usando apenas gadgets obtidos na região de código delimitada pelos símbolos `start_farm` e `mid_farm`.
- O ataque neste nível pode ser feito com apenas dois gadgets.
- Para realizar este ataque você precisa carregar valores novos nos registradores. Pergunte-se: quais dasinstruções descritas acima permitem carregar um novo valor no registrador? Como elas operam?
- Você talvez esteja acostumado a usar o `gdb` no modo `-tui` com o layout asm, que mostra automaticamente o código desmontado. Neste ataque o debugger pode se perder ao lidar com um programa que subitamente executa instruções em endereços fora dos símbolos originais. Uma maneira de forçar o debugger a reinterpretar as instruções é através do comando `disas` (disassemble). O comando `disas $rip,$rip+90` atualiza o quadro de disassemble partindo do contador de instruções atual.
- O comando do gdb `disas` *inicio*, *fim* também pode ser útil para verificar se uma sequência de bytes iniciada em *inicio* efetivamente codifica a instrução que você deseja.

5.2 Fase 5 (ou `rtarget`–*Level 3*)

Antes de prosseguir na Fase 5, vamos parar para ponderar o que foi obtido até agora. Nas fases 2 e 3, você forçou um programa a executar um código que você construiu. Estivesse CTARGET em um servidor na rede, você teria injetado o seu próprio código em uma máquina remota[4]. Na Fase 4 você contornou as duas principais técnicas empregadas por sistemas modernos para impedir ataques de injeção de código. Mesmo sem injetar seu próprio código você foi capaz de forçar um comportamento através da composição de pedaços de código pré-existente. Mais objetivamente, você obteve 95/100 do exercício (ou 9,5). Esta é inegavelmente uma boa nota. Se você tem outras obrigações acadêmicas pendentes, considere parar por aqui. A Fase 5 requer um ataque ROP em RTARGET invocando a função `touch3` com um ponteiro para a string que contém a representação de seu cookie. À primeira vista isso não parece ser significativamente mais difícil do que construir um ROP que invoca `touch2`, mas é! Não se iluda com o valor de 0,5 pontos atribuídos à Fase 5, esta é a fase mais difícil do exercício. Pense nestes 0,5 pontos como um "crédito extra" para os que querem ir além das espectativas normais do curso.

Para resolver a Fase 5, você pode suar gadgets da região delimitada pelos símbolos `start_farm` e `end_farm`. Além das instruções empregadas na Fase 4 esta região inclui codificações de instruções `movl`, como visto na Figura 3C. Além disso este trecho da gadget farm contém também pares de bytes que codificam instruções que não fazem nada (não alteram nenhum valor de registrador ou da memória). Estas instruções, mostradas na Figura 3D, incluem operações como `andb %al, %al`, que operam nos bytes de menor ordem dos registradores, mas não alteram os seus valores.

Sugestões:

- Revise o efeito que uma instrução `movl` tem nos 4 bytes superiores de um registrador.
- A ABI Amd64 SystemV especifica que a pilha deve estar alinhada em 16 bytes, de modo que o valor de `%rsp` seja divisível por 16 no *retorno* de cada função. Você provavelmente violou esta

[4]De certa forma ele está, mas a validação é feita em um ambiente seguro (espero).
–Thiago

restrição algumas vezes nas etapas anteriores sem causar muitos problemas, mas a chamada à função `hexmatch` que verifica o seu exploit talvez falhe se a pilha estiver desalinhada. Caso identifique erros do tipo "segmentation fault" dentro de `hexmatch`, realinhe a pilha adicionando mais um gadget.

- A solução "oficial" é composta de 8 gadgets.

Boa sorte!

A Usando HEX2RAW

HEX2RAW usa como entrada uma string com hexadecimais. Neste formato, calda valor de byte é representado por dois dígitos hexadecimais.

Por exemplo, a string "`012345`" pode ser fornecida no formato hexadecimal como "`30 31 32 33 34 35 00`." (O código ASCII para o dígito decimal x é `0x3`x e o fim de uma string é indicado por um byte nulo.)

Os números hexadecimais que você passa para HEX2RAW devem estar separados por espaços em branco ou quebras de linha. Recomenda-se separar diferentes porções do seu código com quebras de linha enquanto você trabalha nele. HEX2RAW suporta comentários no estilo da linguagem C, de modo que você pode marcar trechos do seu código. Por exemplo:

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

Certifique-se de deixar espaços ao redor dos marcadores de início e fim de comentário ("`/*`", "`*/`"),

Se você gerar um exploit no – por exemplo – arquivo `exploit.txt`, você pode aplicá-lo ao seu programa CTARGET ou RTARGET de diversas formas:

1. Você pode construir uma série de *pipes* (redirecionamento e encadeamento de entradas e saídas de programas) para passar a string por HEX2RAW.

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

2. Você pode armazenar a string processada em um arquivo e usar redireção de I/O:

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

   Esta abordagem pode ser empregada quando executando de dentro do gdb: GDB:

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

3. Você pode também armazenar a string processada em um arquivo e usá-la como um parâmetro na linha de comando::

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

   Esta abordagem também pode ser usada com o GDB.

B Gerando Código de Bytes

Uma maneiera conveniente de se obter Código de Bytes é usar o GCC como um assembler e OBJDUMP como um disassembler. Por exemplo, suponha que você escreva o arquivo `example.s` contendo o seguinte código em 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
```

Este código contém uma combinação de instruções e dados. Tudo à direita de '`#`' 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 desmontado `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 no final mostram o código de máquina gerado a partir das instruções em linguagem de montagem. Cda linha tem um número hexadecimal à sua esquerda indicando o endereço de início de cada instruçõa enquanto que os dígitos hexa após o caractere '`:`' indicam os bytes que codifica a instrução. Assim, pode-se ver que a instrução `push $0xABCDEF` tem o código de Bytes em hexa `68 ef cd ab 00`.

Deste arquivo é simples obter a sequência de bytes para o código:

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

Esta string pode ser passada por HEX2RAW para gerar uma entrada para programas alvo. Alternativamente, você pode editar o arquivo example.d para remover os dados desnecessários e adicionar comentários para legibilidade, produzindo::

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

Esta também é uma entrada válida que pode ser processada por HEX2RAW antes de enviá-la a um dos 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.