第10章 バッファーオーバフロー

10.1 BOF

10.1.1 BOFとは

**BOF（Buffer Over Flow）**または **BOR（Buffer Over Run）**はプログラムのバッファ領域を破壊することによって、メモリ内の値を書き換え、プログラムを誤作動させる手法である。典型的な脅威の一つであるゼロデイは主にこの手法によって引き起こされる。最も有名な手法としては、スタック領域に積まれた関数やサブルーチン（以後最近のプログラミング言語を考慮して関数とのみ記述）のリターンアドレスを上書きしてスタック領域に送り込んだプログラムを実行させる方法（図10.1）があるが、その他にスタック上の変数の値を書き換えて関数の実行を誤らせたり、ヒープ領域からの関数の呼び出しを改変して違う関数を呼び出したりする手法がある（**ヒープオーバーフロー**）。

|  |  |  |  |
| --- | --- | --- | --- |
| メモリ空間 | |  |  |
| バッファ | ⇒ | | バッファ |
| データ |  | |  |
| データ |  | |  |
| レジスタの内容 |  | |  |
| Return Address |  | | Return Address |

図10.1 バッファオーバーフロー

なお、Java言語などではメモリへのアクセスをVM(バーチャルマシン)が管理するため原理的にはBOFを起こさないが、VMがC/C++言語で記述されている場合はVM自体がBOFを起こす可能性がある。

この章では実際のC言語のコードを元にBOFの手法について説明する。ただし実際にBOFを起こすことは難しいので、サンプルプログラムと実行環境には若干の作り込みと制限を加える。処理系としてはOSにLinuxのCentOS7（64bit）、 C/C++言語のコンパイラには gcc/g++ （バージョン4.8.5）を使用する。またCPUアーキテクチャはx64(AMD64, Little Endian)を使用するので、可能ならば x64(AMD64)アーキテクチャのアセンブラの基本的な知識があると理解が一層深まるが必須ではない。

10.1.2 メモリ構造

図10.2 にプログラムのメモリ上の配置の概要を示す（Linuxの場合）。ただしこの構造は処理系（OS）によって異なる。テキスト領域にはプログラムの実行コードが格納され、静的領域には静的変数が格納される。これらの領域はプログラム（ソースコード）のコンパイル時に確保される。一方**ヒープ領域**はプログラム中で動的に確保されるメモリ領域であり、**スタック領域**には関数からのリターンアドレスやローカル（自動）変数などが積まれる（全体の使用可能な領域自体はコンパイル時に確保される）。ここでバッファ領域とはメモリ上の実際の領域ではなく、プログラミング上でのメモリ領域の使用方法を表しているものとする。

|  |  |
| --- | --- |
| テキスト領域 | Low Address  High Address |
| 静的領域 |
| ヒープ領域  スタック領域 |

図10.2プログラムのメモリ上の配置

10.1.3 関数の呼び出しとアドレスマップ

プログラム中からある関数を呼び出す場合、関数の終了時にまた元の呼び出した場所に戻ってくる必要があるために、リターンアドレスをスタックに積んで（さらに必要なデータがある場合はそれらもスタックに積んで）から該当関数に制御を移す。

図10.3にC言語の簡単なサンプル（bof-1.c）とコンパイル・実行の結果を示す。実行結果の "BUF =" (ソースコードではL12)はスタック領域上に確保されたbuf[]の先頭アドレスを示している（この値は毎回変化する）。

図10.4はbof-1.cのオブジェクトモジュールである bof-1.o を作成し、それを逆アセンブルしたものである。実行可能ファイルを逆アセンブルしてもよいが、オブジェクトモジュールの方がライブラリ等をリンクしていない分、見やすい出力となる。

図10.4の5d:行でmain()関数から sum()関数をコールしているが、この時点でリターンアドレスが自動的にスタックに積まれる。sum()関数では0:行でベースポインタ(rbp)レジスタの内容をスタックに退避させ、1:行でその時点でのスタックポインタ(rsp)が示すアドレスをベースポインタ（sum()関数のベースとなるアドレス）としている。また4:行でスタックポインタの値を 0x30（48Byte）減算することによって48Byteのバッファ領域を確保している。続いて 8:行、 c:行で引数が格納されたレジスタの内容（つまり引数）をスタックの上位部分に積んでいる（32bit CPUでは引数は直接スタックに積まれるが、レジスタの容量に余裕のある64bit CPUでは引数はレジスタに格納される）。

つまりこの時点でのプログラムのスタック領域は図10.5のようになる．C言語では配列の添え字に関する指定制限はないので、リターンアドレスはbuf[5]に相当することになる。

10.1.4 リターンアドレスの書き換え

図10.5のメモリマップを踏まえ、bof-2.c のようなプログラムを作成してみる（図10.6）。bof-1.cと比較して dummy()という関数が追加されている．また18行目で dummy()関数の先頭アドレスを buf[5]に格納している。つまり sum関数のリターンアドレスを dummy()関数のアドレスに書き換えてしまっている。このプログラムが実行されると、sum()関数終了時に本来ならば元のmain()関数に制御が移るはずであるが、リターンアドレスが書き換えられてしまっているためdummy()関数に制御が移ってしまう。

もしバッファに格納されるデータとして何らかのプログラムを流し込み、同時にリターンアドレスを書き換えて、流し込んだプログラムに制御を移す事が可能であれば、そのプログラムを乗っ取ることが可能となる。

1 #include <stdio.h>

2 #include <stdlib.h>

3

4 long int sum(long int a, long int b)

5 {

6 long int buf[3];

7

8 buf[0] = a;

9 buf[1] = b;

10 buf[2] = buf[0] + buf[1];

11

12 printf("BUF = %016lx\n", buf);

13 return buf[2];

14 }

15

16 int main(void)

17 {

18 long int plus = sum(1, 2);

19

20 printf("ANS = %ld\n", plus);

21 exit(0);

22 }

$ gcc bof-1.c -o bof-1

$ ./bof-1

BUF = 00007fff7bb1ebd0

ANS = 3

図10.3 bof-1.cとその実行結果

|  |  |
| --- | --- |
| 2 | -0x30 **rsp** |
| 1 | -0x28 |
| buf[0] | -0x20 |
| buf[1] | -0x18 |
| buf[2] | -0x10 |
| (buf[3]) | -0x08 |
| rbp (buf[4]) |  |
| リターンアドレス (buf[5]) |  |

図10.5 スタック領域のアドレスマップ

$ gcc bof-1.c -c

$ objdump -d bof-1.o

0000000000000000 <sum>:

0: 55 push %rbp

1: 48 89 e5 mov %rsp,%rbp

4: 48 83 ec 30 sub $0x30,%rsp

8: 48 89 7d d8 mov %rdi,-0x28(%rbp)

c: 48 89 75 d0 mov %rsi,-0x30(%rbp)

10: 48 8b 45 d8 mov -0x28(%rbp),%rax

14: 48 89 45 e0 mov %rax,-0x20(%rbp)

18: 48 8b 45 d0 mov -0x30(%rbp),%rax

1c: 48 89 45 e8 mov %rax,-0x18(%rbp)

20: 48 8b 55 e0 mov -0x20(%rbp),%rdx

24: 48 8b 45 e8 mov -0x18(%rbp),%rax

28: 48 01 d0 add %rdx,%rax

2b: 48 89 45 f0 mov %rax,-0x10(%rbp)

2f: 48 8d 45 e0 lea -0x20(%rbp),%rax

33: 48 89 c6 mov %rax,%rsi

36: bf 00 00 00 00 mov $0x0,%edi

3b: b8 00 00 00 00 mov $0x0,%eax

40: e8 00 00 00 00 callq 45 <sum+0x45>

45: 48 8b 45 f0 mov -0x10(%rbp),%rax

49: c9 leaveq

4a: c3 retq

000000000000004b <main>:

4b: 55 push %rbp

4c: 48 89 e5 mov %rsp,%rbp

4f: 48 83 ec 10 sub $0x10,%rsp

53: be 02 00 00 00 mov $0x2,%esi

58: bf 01 00 00 00 mov $0x1,%edi

5d: e8 00 00 00 00 callq 62 <main+0x17>

62: 48 89 45 f8 mov %rax,-0x8(%rbp)

66: 48 8b 45 f8 mov -0x8(%rbp),%rax

6a: 48 89 c6 mov %rax,%rsi

6d: bf 00 00 00 00 mov $0x0,%edi

72: b8 00 00 00 00 mov $0x0,%eax

77: e8 00 00 00 00 callq 7c <main+0x31>

7c: bf 00 00 00 00 mov $0x0,%edi

81: e8 00 00 00 00 callq 86 <main+0x3b>

図10.4 bof-1.oとその逆アセンブラ

1 #include <stdio.h>

2 #include <stdlib.h>

3

4 void dummy(void)

5 {

6 printf("Hell World!!\n");

7 exit(1);

8 }

9

10 long int sum(long int a, long int b)

11 {

12 long int buf[3];

13

14 buf[0] = a;

15 buf[1] = b;

16 buf[2] = buf[0] + buf[1];

17

18 buf[5] = (long int)dummy;

19 printf("BUF = %016lx\n", buf);

20 return buf[2];

21 }

22

23 int main(void)

24 {

25 long int plus = sum(1, 2);

26

27 printf("ANS = %ld\n", plus);

28 exit(0);

29 }

$ gcc bof-2.c -o bof-2

$ ./bof-2

BUF = 00007fff7bb1ebd0

Hell World!!

図10.6 bof-2.cとその実行結果

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <string.h>

4

5 #define BUFSZ 64

6

7 void print\_buf(unsigned char\* buf,int size)

8 {

9 int i;

10 for(i=0; i<size; i++) {

11 printf("%02x ", buf[i]);

12 if (i%8==7) printf("\n");

13 }

14 printf("\n");

15 fflush(stdout);

16 }

17

18 // 破壊防止用にグローバル変数化

19 FILE\* fp;

20 int nm;

21 unsigned char cc;

22

23 void bof(void)

24 {

25 unsigned char buf[BUFSZ];

26

27 // バッファの初期化と表示

28 memset(buf, 0, BUFSZ);

29 print\_buf(buf, BUFSZ\*2);

30

31 // アドレス表示

32 printf("DATA = %016lx\n", buf);

33 fflush(stdout);

34

35 // ファイルから読み込み

36 fp = fopen("dat", "r");

37 if (fp!=NULL){

38 nm = 0;

39 cc = fgetc(fp);

40 while(!feof(fp)) {

41 buf[nm++] = cc;

42 cc = fgetc(fp);

43 }

44 fclose(fp);

45 }

46

47 // バッファ表示

48 print\_buf(buf, BUFSZ\*2);

49 return;

50 }

51

52 int main(void)

53 {

54 bof();

55 exit(0);

56 }

図10.7 bof-3.c のソースコード

10.1.5 BOFに対する防御

現在では、リターンアドレスの書き換えに対する防御方法としては以下の対策が取られている。

【ASLR (Address Space Layout Randomization) 】

データ（ヒープ・スタック）の領域をメモリ内にランダムに配置することによって、リターンアドレスを特定できないようにする。

【DEP (Data Execution Prevention)：NX 】

データの領域中に存在するコードの実行を禁止する。

【SSP (Stack Smashing Protector)：スタックカナリア】

スタックの破壊を監視する手法である。監視用のスタックデータを用意し、そのスタックデータが書き換えられた場合に、スタックが破壊されたとしてプログラムの実行を中断する。ただしこの機能はデフォルトでOFFとなっている。有効にするには -fstack-protector オプション付きでコンパイルする必要がある。

スタックカナリアという名称は、かつて鉱山での有毒ガスの検出にカナリアが利用されていたことに由来する。

【PIE (Position Independent Excutable)：位置独立実行形式】

データ領域だけで無く、実行プログラム本体のテキスト領域（実行コードが保存される領域）のアドレスも可変にすることによって、プログラム全体を位置独立とする（-fpie オプション付きでコンパイルする）。ただしこの機能はデフォルトでOFFとなっている。

一方共有ライブラリのテキスト領域のアドレス（のみ）を可変にする場合は -fpic オプション付きでコンパイルする（**PIC**: Postion Independent Code：位置独立コード）。