恶意代码分析

第二章:漏洞类型与挖掘技术

大纲

- 漏洞类型与挖掘技术
  - 漏洞类型
  - 漏洞挖掘方法介绍
  - 漏洞快速定位技巧
  - 漏洞审计案例
  - 模糊测试
  - 基于反馈的模糊测试工具 AFL

漏洞类型

- 逻辑错误
- 内存破坏(重点关注)
  - 栈溢出
  - 整数溢出
  - 格式化字符串
  - 堆溢出
  - 释放后使用

栈溢出（Stack Overflow）

In one contemporary operating system, one of the functions provided is to move limited amounts of information between system and user space. The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be overlaid by the user. This can be used to inject code into the monitor that will permit the user to seize control of the machine.

- 栈溢出的历史
  - 概念提出于美国空军发表的论文《Computer Security Technology Planning Study》(1972)
  - Morris 蠕虫 (1988)
  - 发表在Phrack杂志上的利用技术文章《Smashing the Stack for Fun and Profit》(Aleph One 1996)

栈溢出（Stack Overflow）

- 文章《Smashing the Stack for Fun and Profit》中的案例

```
void function(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```

整数溢出（Integer Overflow）

- 历史
  - 1996年，欧洲航天局阿丽亚娜5号火箭爆炸，事故原因就是整数溢出
  - M. Dowd, C. Spencer, N. Metha, N. Herath, and H. Flake. Advanced Software Vulnerability Assessment. In Blackhat USA, 2002年8月
    - IO2BO（Integer Overflow to Buffer Overflow）
    - 数组下标越界

IO2BO

```
void safe_memcpy(char *src, int size) {
    char dst[512];
    if (size < 512) {
        memcpy(dst, src, size);
    }
}
```

数组下标越界

```
  void safe_set_element(char *arr, int
index, char value, int arr_size) {
      if (index < arr_size) {
          arr[index] = value;
      }
  }
```

格式化字符串（Format String）

- 历史
  - 发现于1999年
  - 《Format String Attacks》, Newsham (2001)
    - 利用占位符%x和%n来实现任意内存读写

不安全的代码

```
int func(char *user) {
     printf(user);
}
```

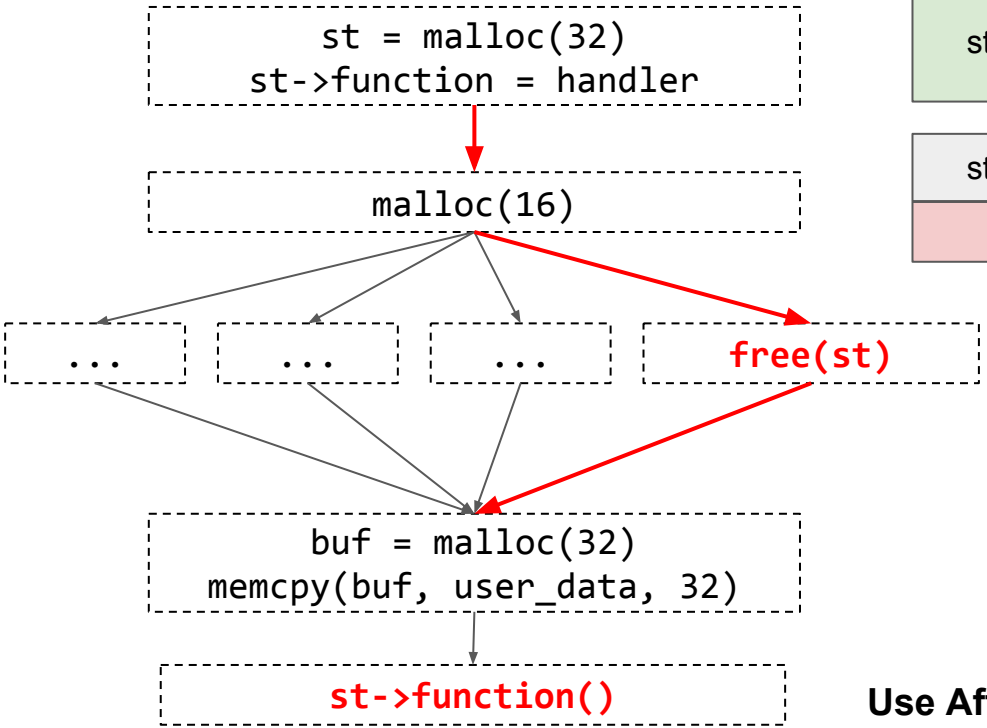
安全的代码

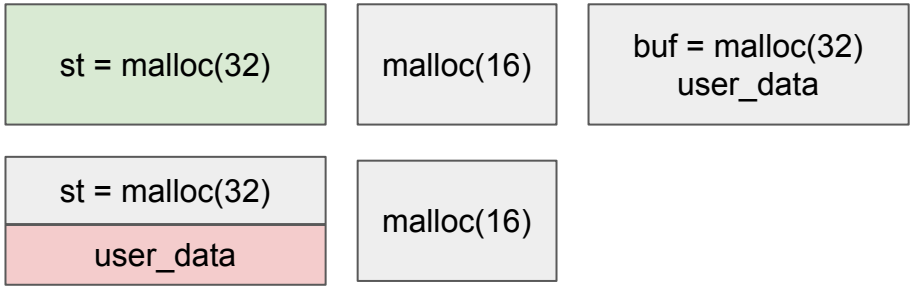
```
int func(char *user) {
     printf("%s", user);
}
```

可能存在格式化字符串漏洞的函数

- fprintf - prints to a FILE stream
- printf - prints to the ‘stdout’ stream
- sprintf - prints into a string
- snprintf - prints into a string with length checking
- vfprintf - print to a FILE stream from a va_arg structure
- vprintf - prints to ‘stdout’ from a va_arg structure
- vsprintf - prints to a string from a va_arg structure
- vsnprintf - prints to a string with length checking from a va_arg structure
- setproctitle - set argv[]
- syslog - output to the syslog facility

释放后使用（Use After Free）





结构/对象在释放（free/delete）后，其引用依然错误地被使用，我们称这种情形为释放后使用（Use After Free）

左图的例子中，结构st在某代码路径下被释放，之后其函数指针又被调用，出现了释放后使用的漏洞。

Use After Free!

堆溢出（Heap Overflow）

- 历史
  - 《Vudo Malloc Tricks》, 2001
  - 《Once Upon A free()》, 2001
  - 《The Malloc Maleficarum》, 2005
  - 《MALLOC DES-MALEFICARUM》, 2009
  - 自2014年hack.lu CTF赛题oreo以来，越来越多的堆利用技巧被 发明

```
#define BUFSIZE 256
int main(int argc, char **argv) {
     char *buf;
     buf = (char*)malloc(sizeof(char)*BUFSIZE);
     strcpy(buf, argv[1]);
}
```

漏洞挖掘与利用一般流程

1. 信息收集
    a. 文档
    b. 历史公开漏洞
    c. 代码/软件
    d. 软件依赖
2. 攻击面分析
    a. 整理攻击者可以触及的相关模块/代码
    b. 理解与目标软件的交互方法
3. 寻找漏洞/崩溃（Crash）
    a. 编写触发漏洞POC
    b. 编写漏洞利用（Exploit）

发现漏洞的核心方法

- 静态分析(Static Analysis)- 气宗
  - 代码审计
    - 手动
    - 自动:Coverity/Fortify
  - 逆向分析
- 动态分析(Dynamic Analysis)- 剑宗
  - 模糊测试 Fuzzing
    - Google的ClusterFuzz
  - 符号执行(学术研究阶段)
    - klee/s2e/angr

代码审计/逆向分析

- 定位攻击面相关代码/输入处理代码
  - 查找相关系统API调用
  - 通过关键词匹配
- 肉眼定位漏洞
  - 可以按照漏洞模式匹配
  - 取决于对目标软件理解程度
  - 需要经验积累

举例: 审计服务端软件

- 定位攻击面相关代码/输入处理代码
  - TCP
    - 查找listen函数调用
    - 查找recv函数调用
  - UDP
    - 查找recvfrom函数调用
  - 查找协议中的关键词
    - HTTP协议：GET/POST/HTTP 1.1/Accept/Authorization/Cookie
    - UPNP SSDP协议：M-SEARCH/ssdp:discover

按照漏洞类型快速定位漏洞

- 缓冲区溢出（堆溢出/栈溢出）
  - grep（或使用IDA中的快捷键x）寻找危险函数调用，例如strcpy/sprintf/sscanf/...
  - 检查内存拷贝函数（memcpy/strncpy/snprintf/...）中的长度参数是否根据目 标来指定
  - 关注循环中的赋值
- 整数溢出（Integer Overflow）
  - 检查代码对输入（例如协议中的整数字段）中的整数 处理
  - 有符号无符号、默 认类型转换
- 格式化字符串（Format String）
  - grep（或使用IDA中的快捷键x）寻找并检查相关函数调用，例如printf/sprintf/snprintf/...
- 释放后使用/多次释放（Use After Free/Double Free）
  - 基于全局的理解
  - 关注对象引用的分配和 释放

缓冲区溢出案例(某IoT设备)

```
int v19; // [sp+30h] [bp-7C98h]@10
int v20; // [sp+767Ch] [bp-64Ch]@3
int buf; // [sp+7C68h] [bp-60h]@5

fd_ = fd;
para_buf_ = para_buf;
para_len_ = para_len;
total_len_ = total_len;
v11 = 'PTTH';
v12 = '0.1/';
v13 = '002 ';
v14 = '\rKO ';
v15 = '{\n\r\n';
v16 = 'rats';
v17 = '"":t';
v18 = ',';
if ( dword_32E4C || (map_somefile("/home/mmap_tmpfs/mmap
{
  if ( dword_32BA8 == -1 && sub_133B0() )
  {
    v8 = (char *)&v18 + sprintf((char *)&v18 + 1, "ret_c
    v9 = v8 + 7;
  }
  else if ( read_if_there_is_more(fd_, para_buf_, para_len_
  {
    if ( check_para("off=", &buf, (const char *)&v20) )
```

```
char *__fastcall check_para(const char *para_off, void *dest, const char *a3)
{
  const char *v3; // r6@1
  const char *para_off_; // r4@1
  void *dest_2; // r5@1
  char *result; // r0@1
  const char *v7; // r6@1
```

```
    if ( !strcmp(v11, "key_input=") )
    {
      memset(&v17, 0, 0x100u);
      memcpy(&v17, v12, len);
      v14 = sub_A8A0((const char *)&v17);
      v15 = v14;
      v16 = strlen(v14);
      memcpy(dest_2, v15, v16);
      result = (char *)1;
    }
    else
    {
      memcpy(dest_2, v12, len);
      result = (char *)1;
    }
  }
  return result;
```

重点关注内存拷贝函数的用法。

此处 memcpy 的长度按照源(source)来指定，因此存在溢出。

缓冲区溢出案例

```
int read_line(char *p, size_t len)
{
    if (len <= 0) return 0;
    int i;
    for (i = 0; i < len; ++i)
    {
        int ret = read(0, &p[i], 1);
        if (ret <= 0 || p[i] == '\n')
        {
            break;
        }
    }
    p[i] = 0;
    return i;
}
```

对于循环中的赋值，特别注意检查循环终止的条件。

左边函数功能是接收一行输入，存入长度为len的字符串p当中。

当输入长度>=len时，for循环会在i = len时终止，此时程序会往p[i]即p[len]中写入0，发生1个字节的溢出。

整数溢出案例

```
static int perf_swevent_init(struct perf_event*event)
{
    int event_id = event->attr.config; //dangerous!

    ...

    if (event_id >= PERF_COUNT_SW_MAX)
        return -ENOENT;

    if (!event->parent) {
        int err;

        err = swevent_hlist_get(event);
        if (err)
            return err;

        atomic_inc(&perf_swevent_enabled[event_id]);
        event->destroy = sw_perf_event_destroy;
    }

    ...
}
```

左边为Linux内核 CVE-2013-2094漏洞。

event_id用户可任意传入，代码中被定义为有符号的int类型，检查时只检查了上界没有检查负数类型，最终导致被作为数组下标时产生下溢。

重点关注用户输入的整数，其类型的使用、发生运算情况。尤其注意整数被作 为数组下标、缓冲区长度。

格式化字符串案例

https://github.com/php/php-src/commit/b101a6bbd4f2181c360bd38e7683df4a03cba83e:

```
      zend_vspprintf(&message, 0, format, va);
      if (fetch_type & ZEND_FETCH_CLASS_EXCEPTION) {
-           zend_throw_error(exception_ce, message);
+           zend_throw_error(exception_ce, "%s", message);
      } else {
           zend_error(E_ERROR, "%s", message);
      }
```

左边为PHP CVE-2015-8617漏洞。

由于编译器的检查，类似printf这种传统的格式化字符串函数已 经很难出现漏洞。

如果自己包装的格式化字符串函数，也 许会出问题，例如左边PHP的zend_throw_error()。

定位很容易，只需追踪相关 调用，检查格式字符是否可控。

释放后使用案例

```
char* ptr = (char*)malloc (SIZE);
if (err) {
    abrt = 1;
    free(ptr);
}
...
if (abrt) {
    logError("operation aborted before commit", ptr);
}
```

左边简单的案例展示了ptr在某些条件下被释放，同时后续又被使用所产生的UAF漏洞。

一般来说，在真实软件中，例如浏览器，很多结构、对象的操作非常复杂，分支路径甚多，能触发UAF的可能只有少数几条路径，因此想要肉眼找到这些罕见的路径，必须充分理解所有代码分支。

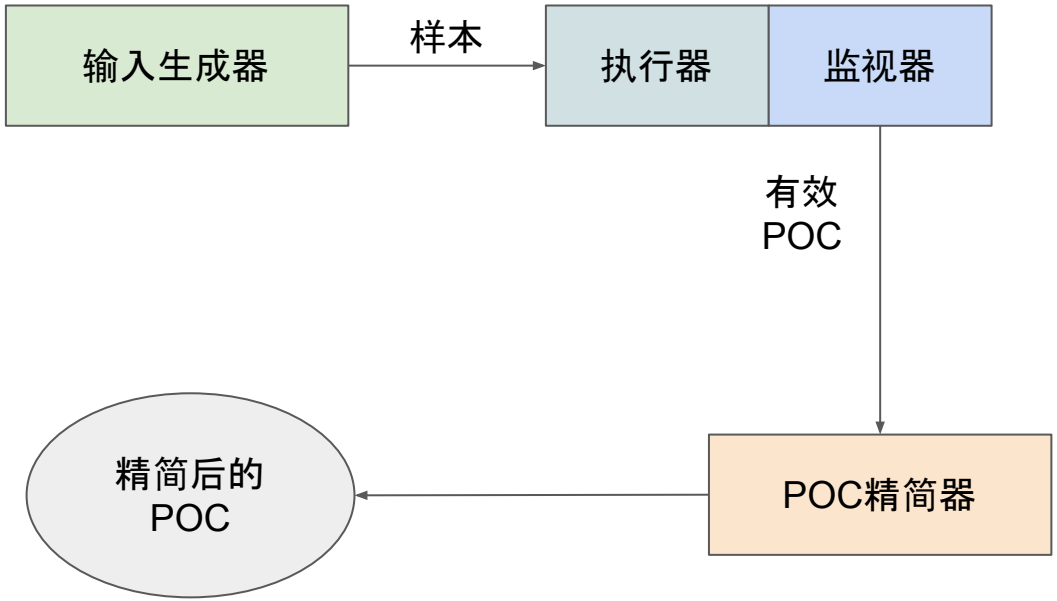
采用代码审计/逆向分析的方法肉眼找UAF的漏洞对研究人员的要求非常高，目前大部分UAF漏洞是通过模糊测试发现的。

高产量漏洞发现方法:模糊测试

- 什么是模糊测试
  - 一种通过向目标系统提供大量非预期的输入并监视异常结果来发现软件漏洞的方法
- 模糊测试流程
  - 确定测试目标
  - 生成输入样本
  - 传递给目标软件进行执行
  - 监控软件状态

如何编写自己的Fuzzer

- 样本输入生成、变异模块
  - 文件格式：图片、字体、文档
  - 脚本：JavaScript、ActionScript
  - 协议：HTTP、FTP、SMB
- 执行和监控模块
  - 调试器捕捉崩溃
- POC精简模块
  - 尝试缩小样本，看是否能够继续使目标崩溃，直到无法缩小



举例：Fuzz Web服务器

- 样本输入生成、变异模块
  - 生成符合HTTP协议的畸形HTTP数据包
  - 覆盖尽可能多的HTTP选项

```
POST / HTTP/1.1
Accept-Encoding: identity
Content-Length: 31337
Host: 127.0.0.1
Connection: '(834j777fjads""""""
Keep-Alive: 115
Accept-Charset:
AAA...AA%s%s%s%s%s%s%s%s%s%s%s%s%n%n%n%n%n%n%
n%n%n
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686;
rv:36.0) Gecko/20100101 Firefox/36.0
```

```
ACL / HTTP/1.1
Accept-Encoding: identity
Content-Length: 0
Host: 127.0.0.1
Connection: keep-alive
Keep-Alive: 115
Accept:
AAA...AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686;
rv:36.0) Gecko/20100101 Firefox/36.0
```

举例：Fuzz Web服务器

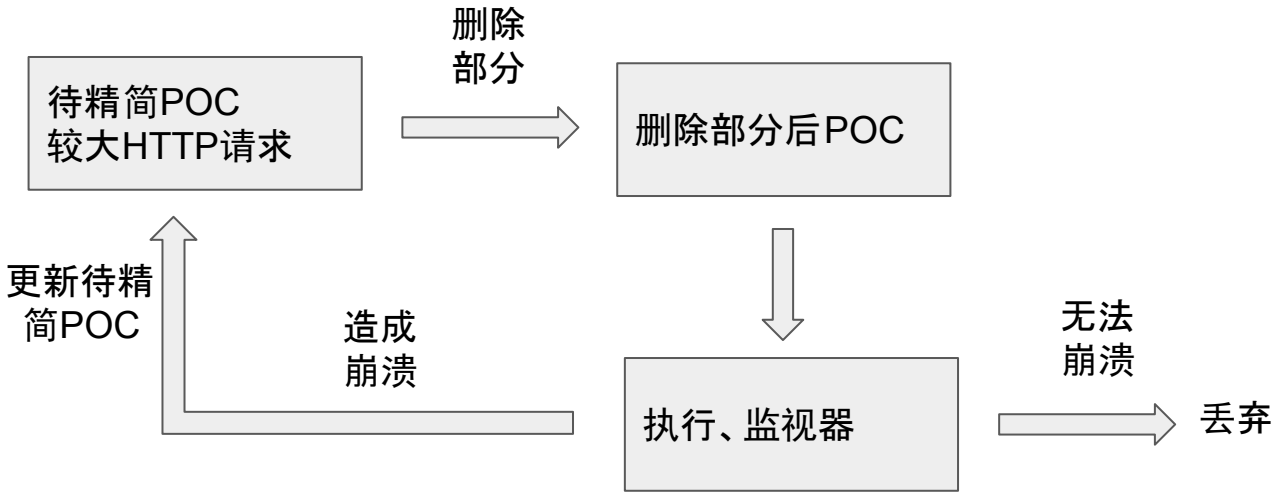
- 软件状态监控
  - 使用python-ptrace库实现一个调试器，来对进程状态进行监控
  - 一旦检测到崩溃，记录崩溃现场和样本信息

```
debugger = ptrace.debugger.PtraceDebugger()
process = debugger.addProcess(pid, False)
process.cont()
process.waitSignals(signal.SIGSEGV)
print "[*] Crash detected"
try:
   rip = process.getreg('rip')
   maps = process.readMappings()
   map_ = find_map(maps, rip)
   md5hash = hash_object.hexdigest()
   instr = process.disassembleOne()
   result = '************* SIGSEGV *************\n'
   result += "MAPS: %s\n" % map_
   result += '----------------------------------------\n'
   result += dump_regs(process.getregs())
   result += '\n\n'
   print result
except:
   ...
```

POC精简器

尝试缩小样本，看是否能够继续使目标崩溃，直到无法缩小。

可设定阈值，例如尝试n次不成功即可终止精简。

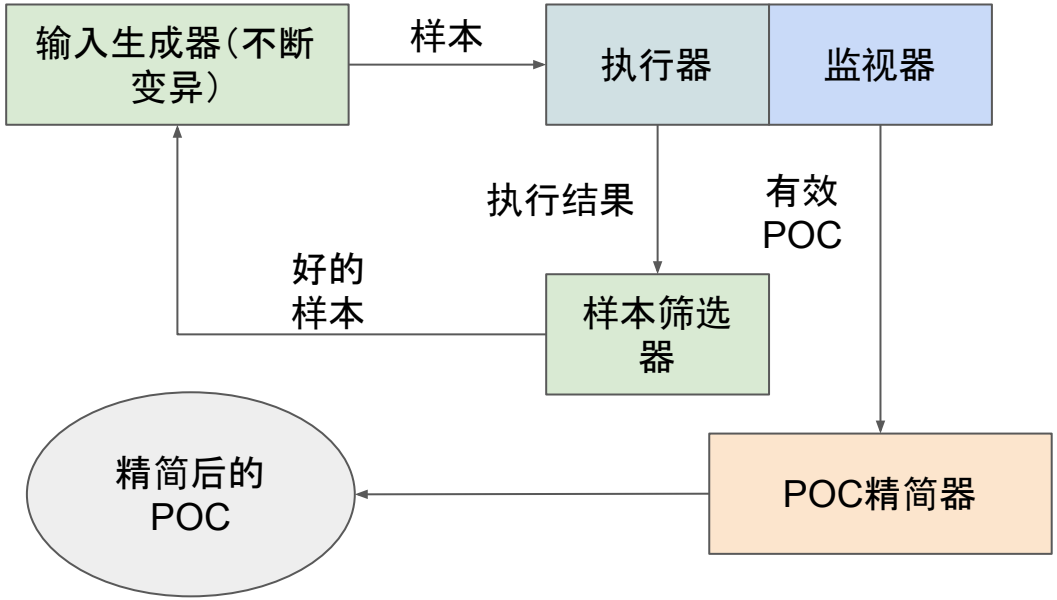


如何评价模糊测试的效果？

- Fuzz速度
  - 单位时间内测试目标处理输入的数量
  - 优化的方案可节约大量计算资源
- 代码覆盖率
  - 基于Fuzzer生成的样本，目标软件所能执行的代码占总代码的比例
  - 测试覆盖的代码越多，效果约好
  - 如何优化？基于反馈的模糊测试

优化思路:基于反馈的模糊测试方法

- 带信息输出的执行器
  - 代码覆盖率
  - 基本块执行记录
- 样本筛选器
  - 根据执行结果，剔除代码覆盖较差的样本
  - 提高样本生成和变异的效率
- 开源工具
  - AFL



American fuzzy lop

- 模糊测试（Fuzzing）
  - 常用黑盒测试技术
  - 构造恶意输入
  - 寻找崩溃
- AFL is unique
  - 基于遗传算法: 利用分支跳转信息（灰盒）
  - 擅长发现深度路径
  - 已发现上百漏洞
- 开源软件 by Michal Zalewski (lcamtuf)
  - http://lcamtuf.coredump.cx/afl/

使用 AFL

- fuzz开源软件
  - 用 afl-gcc/afl-g++ 编译源码
  - 通过LLVM pass进行代码插桩（Instrumentation）
- fuzz二进制软件
  - 使用Qemu 模拟器运行(afl-fuzz -Q 模式)
  - 利用Qemu的动态二进制翻译来进行代码插桩（Instrumentation）
  - 借助Qemu获得多架构支持
  - 速度较慢
- 利用初始样本（Seeds）
  - 以好的初始样本作为种子能加快漏洞的发现

样例:使用 AFL 发现栈溢出

```
$ echo hello > in/hello
$ afl-fuzz -Q -i in -o out ./ropasaurusrex_x64
...
$ ls -l out/crashes/
   -rw------- 1 user user 142 May  8 17:14 id:000000,sig:11,src:000000,op:havoc,rep:64
```