X'CON 2003

缓冲区溢出漏洞发掘模型

作者：funnywei

日期：2003.12

- 内容摘要
  - 简介（**Introduction**）
  - 相关工作（**Related Work**）
  - 我们的模型（**Our Model**）
  - 总结（**Conclusion**）

- 简介
  - 研究的必要性
  - C和C++语言仍然是开发的主要工具
  - 缓冲区溢出攻击已成为主要攻击手段

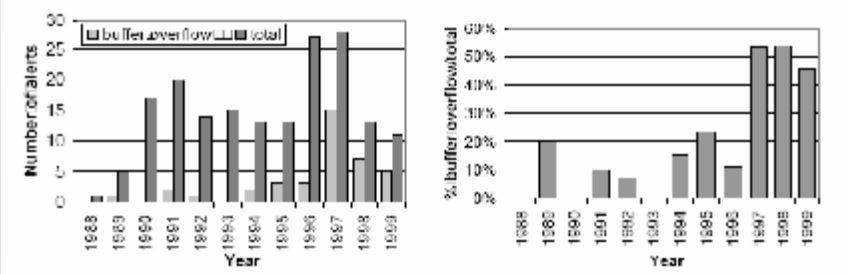
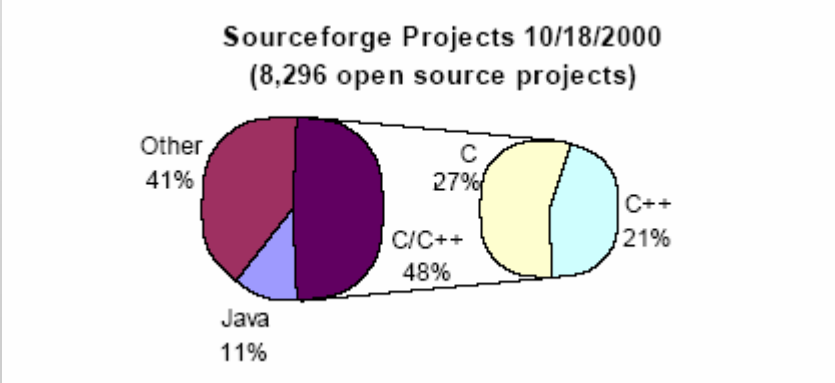


Figure 1: Number of Reported CERT Security Advisories Attributable to Buffer Overflow (Data from [26])



- 相关工作（**Related Work**）
  - 静态检测（**Static Detection**）
  - 动态检测（**Dynamic Detection**）

  主要分为动态和静态检测两种方法，其中静态检测工具主要针对源码做相应检测，而动态检测工具主要从对二进制程序运行时保护的角度出发。

- 静态检测工具
  - 第一代：**lint**
  - 第二代：**Splint**和**LClint**
  - 第三代：**Flowfinder**，**RATS**，**ITS4**

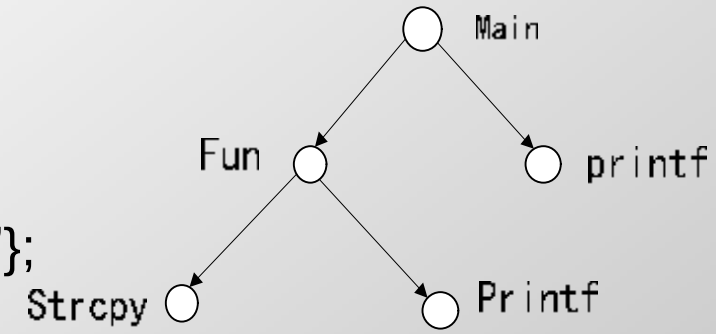
- 动态检测工具
  - **FIST（Fault Injection Security Tool）**
  - **Libsafe和Libverify**
  - **Stack Shield**

- 我们的检测模型
  - 静态分析
  - 动态分析和测试

- 静态分析
  - 目的：得到子过程调用的关系图，以便后续的分析。每一个子过程作为图中的一个节点，同时不安全的函数也作为一个节点存在。
  - 方法：
    - **call**和**ret**指令搜索
    - 编译器优化和特征码匹配
    - 缓冲区边界的定位
  - 辅助：基于污点传播的双向数据流分析

有向图举例：

```
fun(char * arg)
{
    char src[100] = {“This is a test”};
    strcpy(arg,  src);
    printf(“%s \n”, arg);
}
main（）
{
    char dest[100];
    fun(dest);
    printf(“%s\n”, dest);
}
```



- 有向图节点记录
  - 子函数的起始位置
  - 函数所分配的堆栈大小
  - 函数局部变量的使用情况
  - 调用者传递给函数的参数
  - 调用者调用本函数的地址（即函数调用的返回地址）

- **call**和**ret**指令搜索
  - 采用深度优先或者广度优先的搜索算法。

- 编译器优化和特征码匹配
  - 编译器对程序进行了优化以提高执行速度，有些函数（如**strcpy**，**strcat**）被硬编码到程序中。
  - 如：**strcpy**（）

```
main（）
{
   char dest[100]；
   char src[100]；
   gets（src）；
   strcpy（dest，src）；
   return；
}
```

- **VC 6.0编译器**
- **思想：只要在程序中以ECX为核心对指令进行匹配，同时配合这三条字符串操作指令，就可以定位strcat和strcpy。**

```
repne scas  byte ptr [edi]  ；扫描源字符串，长度存放在ecx中
not   ecx
.....
mov   EnX, ECX
.....
shr   ecx, 2
rep   movs    dword ptr [edi],dword ptr [esi]
mov  ecx,      Enx
and   ecx,  3
.....
rep movs    byte ptr [edi],byte ptr [esi]
```

- **VC 7.0编译器**
- strcpy函数也会被硬编码到程序中去，但是它产生的二进制码却和VC6.0产生的二进制码有很大的区别。

strcpy硬编码如下：

```
00401041  mov       cl,byte ptr [esp+eax]
00401045  mov       byte ptr [esp+eax+14h],cl
00401049  inc       eax
0040104A  test      cl,cl
0040104C  jne       00401041
```

因为在这些硬编码的代码中使用了相对基址变址寻址方式，所以在VC7.0下简单提取strcpy的固定二进制特征码比较难，但是我们可以结合指令的语义进行分析，来构建strcpy函数的特征码。

- **缓冲区边界的定位**

  Libsafe对缓冲区估算采用最大化到栈帧的方法。

  本文思想：根据指令对内存单元访问情况来界定缓冲区边界。

  方法：

  （**1**）通过分析未直接访问的堆栈单元来确定目标缓冲区长度

  （**2**）根据局部变量的访问方式来确定目标缓冲区的长度

  （**3**）通过其他方法判断缓冲区长度

- 通过分析未直接访问的堆栈单元来确定目标缓冲区长度
  - 对于普通的变量，如果没有在程序中被使用，那么程序将不会为其在堆栈中保留地址空间。但是对于数组字符数组来说，只要数组中有一个成员被访问，程序就会在堆栈中为整个数组保留地址空间。然而在编译后的程序中没有任何一条指令对其余的地址空间进行访问。

```
int main(int argc, char* argv[]){
char dest[100] = {“this is me”};
printf("%s", dest);
strcpy(dest, "this is a test");
return 0;}
```

首先， sub esp,64h 在堆栈中保留0x64个字节的空间。其次，根据特征码匹配和非安全函数调用匹配找到字符缓冲区的起始位置：

```
0040101E 56                push    esi
0040101F 57                push    edi
    ...
0040105D 8D 54 24 08       lea     edx,[esp+8]
    ...
00401065 8B FA             mov     edi,edx
    ...
0040106A F3 A5   rep movs    dword ptr [edi],dword ptr [esi]
```

由此，我们可以认为字符缓冲区的起始位置为esp+8。由于有压栈的操作，ESP的值不固定，我们用base来代表程序开始时sub esp,64h所指向的位置。esp+8所指向的位置其实就是base。

然后可以分析以ESP,EBP为基址寄存器的指令：

```
00401015 89 44 24 00        mov       dword ptr [esp],eax
0040101E 56                 push      esi
0040101F 57                 push      edi
00401020 89 4C 24 0C        mov       dword ptr [esp+0Ch],ecx
00401024 88 44 24 12        mov       byte ptr [esp+12h],al
00401033 66 89 54 24 10     mov       word ptr [esp+10h],dx
```

由上可以看出在程序保留的变量中，有很大一部分都没有被指令所访问，只是通过rep stos dword ptr [edi]将其设为0，而通过前面的分析已经知道对于普通变量来说，如果没有被指令直接访问就不会为其在堆栈中保留地址空间，所以，我们就可以认为，这些地址空间都在字符数组中。

- **根据局部变量的访问方式来确定目标缓冲区的长度**

```
main（）
{
    char dest[36] = {“this is a test”};
    int a = 1;
    printf（“%s，%d”，dest， &a）；
}
```

对于普通变量来说，如果只是作为目的操作数而从来没有作为源操作数使用，那么它将肯定被编译器优化掉。但是对于数组来说，即使其某个成员从头到尾都作为目的操作数而存在，它也不会被编译器优化掉。通过这个特性，我们就可以确定缓冲区的边界。

在这个程序的汇编代码中没有出现rep stos指令，字符数组的初始化都是通过mov指令来完成的，所以，我们不能通过查找没有直接使用的地址来判断缓冲区的长度。在这个程序中，只有两条指令用堆栈中的变量作为源操作数，

0040103B 8D 54 24 00　　lea　edx,[esp] ；取变量a地址

....

00401043 8D 44 24 04　lea　eax,[esp+4] ；取字符串首址

对于堆栈中其他的地址空间来说，它们都没有被用作源操作数，所以根据我们对编译器优化局部变量原则的分析，我们可以认为这些地址所指向的存储单元都位于字符缓冲区中。

- **通过其他方法判断缓冲区长度**

  有些程序来说，在分配了缓冲区后首先调用函数对其进行初始化 。只要找到相应的API函数调用就很容易判断出目标缓冲区的大小和起始位置。

```
char dest[100];
...
ZeroMemory（dest， 100）；
...
```

  memset()函数在**Release**版本的程序中会被编译成一条汇编语句 **rep stos** 。

- 基于污点传播的双向数据流分析
  - 思想 ： 来源于**Perl**中称为"**tainting**"的机制。监控所谓的"不可信数据"，通常是由用户输入的，将这些输入标记为"**tainted**"。污点数据通过程序语句进行传播。
  - 污点来源：环境变量，参数，文件，网络输入
  - 污点传播在源码检测中的运用
  - 在二进制代码中的应用－－双向数据流分析方法

- **污点传播在源码检测中的运用**
  - 在数据类型前，加一个tainted类型限定词表示该数据来源于一个不可信数据源。例如：
  - int main(int argc, tainted char *argv[]);

  例：wuftpd 2.6.0的格式化字符串漏洞。

  |----site_exec

  |----lreply

  |----vreply

简化看作：

```
while(fgets(buf, sizeof buf, f)){
    lreply(200, buf);}
void lreply(int n, char *fmt, ...){
...
vsnprintf(buf, sizeof buf, fmt, ap);
...}
```

现实中format string的例子并不太可能是显式的使用prinft(buf)这么容易检测。而且多数情况下，真正出错的地方很有可能与最终调用vsnprinft，prinft的函数并不在一个源文件中。例如，上面所举的wuftp“site exec”例子，其分别在ftpcmd.y文件中ftpd.c文件中。

- **偏序关系**

考虑例子：

void f(tainted int);
untainted int a;
f(a)
f期望的是tainted数据，而传入的是untainted的参数。允许！

考虑第二个例子：

void g(untainted int)；
tainted int b;
g(b);

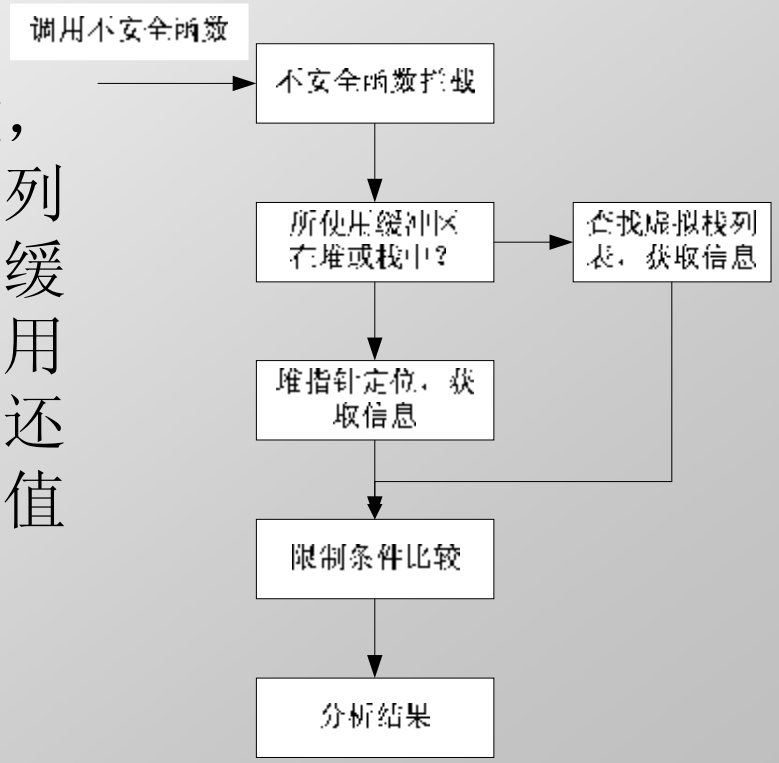
在这种情况下，g期望untainted 而得到tainted的数据，不允许！

把两个例子结合起来，我们得到下面的偏序关系：

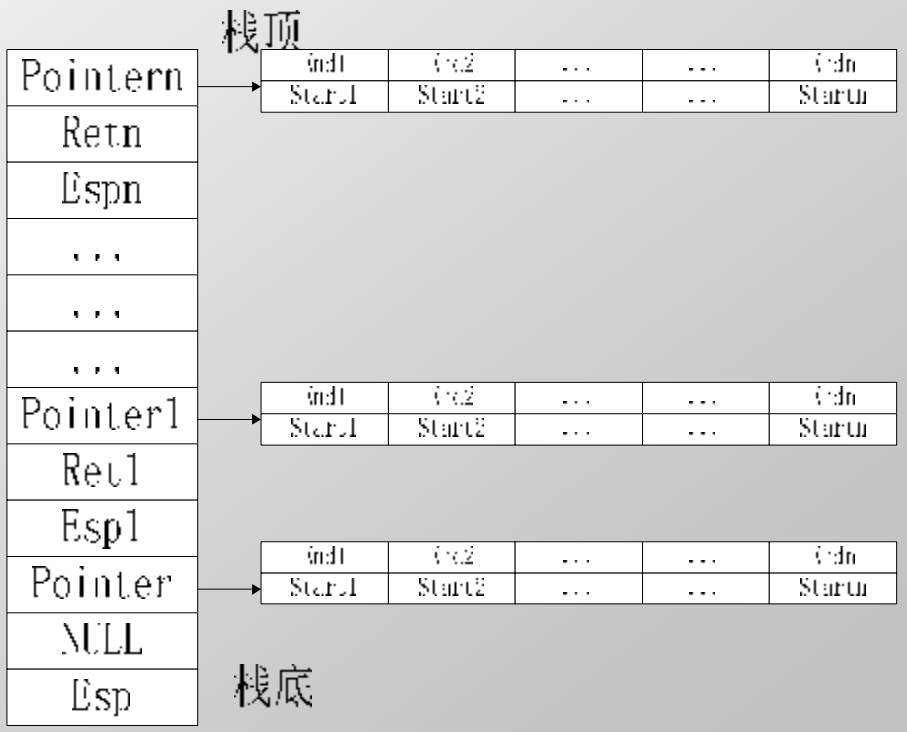
untainted int < tainted int

- 在二进制代码中的应用——双向数据流分析方法
  - 一是从污点数据流开始分析，自上而下形成一棵传播树（Propagation Tree）。二是从strcpy等可能出错的函数开始，向上回溯。找到两者相交的位置，从而产生污染流路径（Tainted Flow Path）。当然相交的位置可能不止一处，那么将产生多条污染流路径。

- 动态分析及测试过程
- **思想：**动态拦截所有可疑的函数，在拦截时创建当前调用的虚拟栈列表。在栈列表中，记录所有函数缓冲区的情况。对于可疑函数所使用的缓冲区，先分析它是位于堆中还是在栈中。接着，获得相应的数值化描述信息，最后与限制条件相比，得到分析结果。



- **虚拟栈**
- 虚拟栈用来记录栈的调用情况。同样，在分析的过程中还可以很方便的定位出所分析的字符缓冲区在堆栈中的位置，生存周期等。

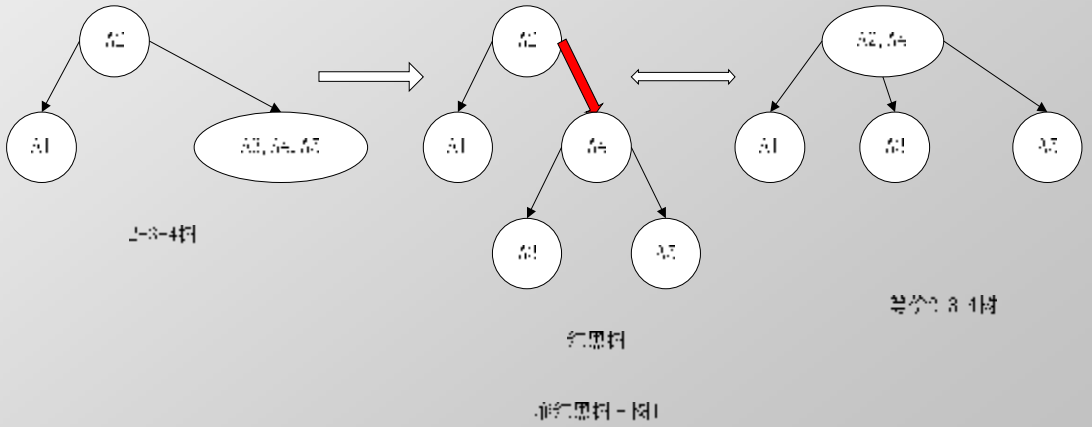


- 缓冲区数值化技术及限制条件的产生
  - **David Wagner**等人在参考文献**[1]**中将缓冲区溢出的检测问题规范化为整数限制的问题，并定义了限制语言（**Constraint Language**）。

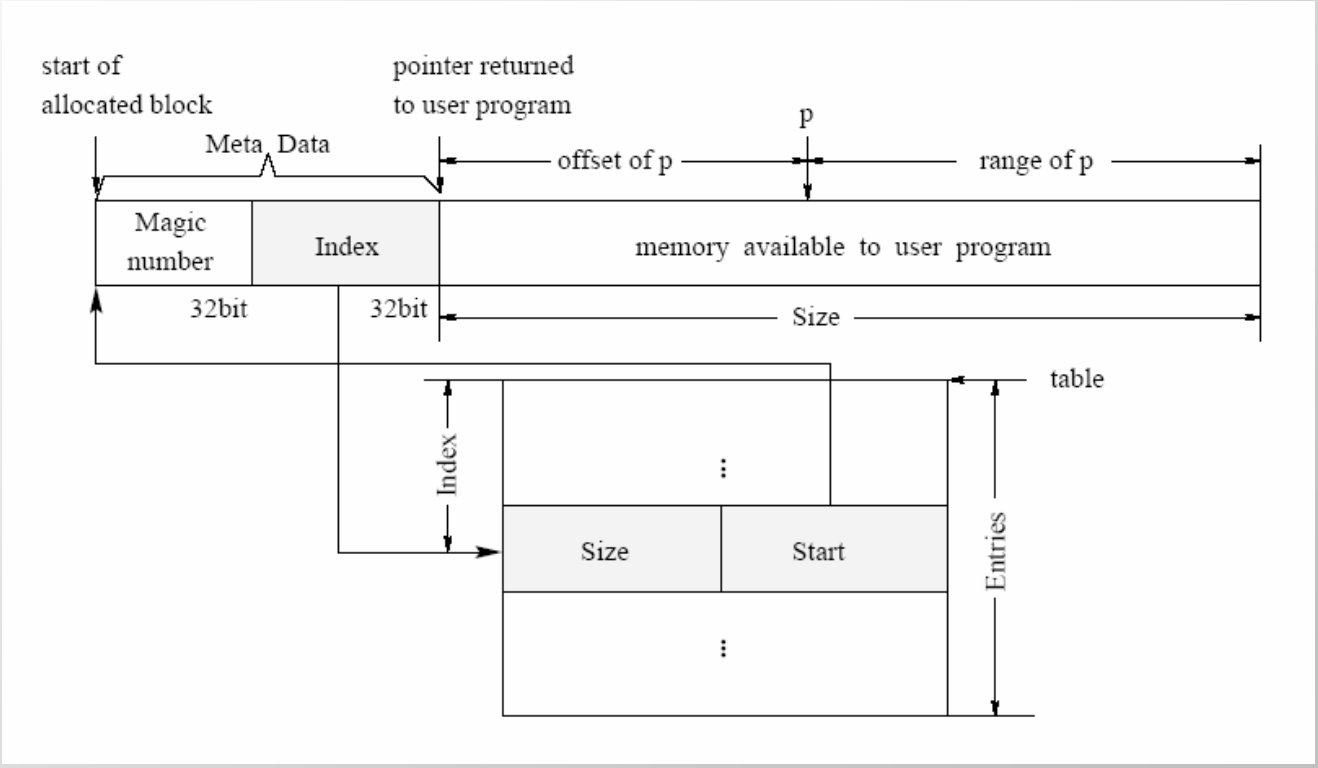
- 限制条件的产生
  - 符号**len(s)**表示当前使用的长度（包含结束字符'**\0**'），范围属性为**[a, b]**。**Alloc(s)**表示**buffer**实际分配的大小,范围属性为**[c, d]**。
  - 在对所有的变量进行了范围推断之后，再进行安全属性检查：
    - 如果**b<c**，溢出不可能发生。
    - 如果**a>d**，那么肯定发生。
    - 如果**d>b>c>a,** 那么溢出有可能发生。

- **堆指针检测的红黑树结构及魔数定位法**
  - 思想：拦截堆内存的分配函数（例如，**Windows**下拦截**RtlHeapAlloc**，**Linux**下拦截**malloc**）。在每次成功分配堆内存的时候，采用红黑树记录堆指针，或者在刚分配的堆内存的起始位置记录**Meta Data**信息。

- **红黑树（Red/Black Tree）检测**
- 树中的每个节点对应于一个已分配内存，其关键字包含block的起始和结束地址。其插入和删除的时间复杂度为O(logn)，采用红黑树而不采用二叉树主要是为了避免最坏情况的产生。导致时间复杂度为O(n)。
- 例子：**addr(A1)<addr(A2)<…<addr(A5)**



- 魔数定位法（**Magic Number Location**）
  - 思想：使用一个特殊的数据结构记录在动态内存的开始地址，叫做 **Meta Data**。采用魔术定位法可以减少插入和删除的复杂度。其搜索时间为**O(1)**，即线性时间。



- 在每个分配的动态内存块的开始部分插入一个头部，来存储特定的元数据。元数据包含两个部分，一个是魔数，一个是表的索引值。魔数是元数据的第一部分，它是精心选择的一个数，在正常的用户程序里不太可能出现。表的每条记录包含了已分配块的地址和大小。
- 对于指针p，搜索其前面的内存来匹配魔数。如果没有匹配，就不可能是一个内存块的起始位置。反之，如果匹配，那么就有可能是我们找到了起始点。当然，很有可能魔数在用户区域里巧合。在魔数相匹配后，再根据index查找表中的相应位置存储的Start值来比较起始地址是不是和分配块的地址一致。

- 进程跟踪与调试
  - 捕获函数调用异常（对于Windows而言）
    - **DebugActiveProcess attach**到一个活动进程。终止调试使用**DebugActiveProcessStop**。任何时候，被调试的进程发生发生异常都会产生**EXCEPTION_DEBUG_EVENT**事件，可能的异常包括试图访问不可访问的内存，执行断点指令，试图被**0**除，或者任何其他**SEH**处理的异常。
  - Ptrace跟踪，LKM拦截异常（对于Linux）
    - *NIX系统下可以采用可加载内核模块（LKM）拦截系统的异常中断。具体方法可以参考Phrack 61的Hijacking Linux Page Fault Handler Exception Table。

- 发生异常时的现场数据保护
  - 现场记录的信息包括：线程上下文、程序加载的模块信息、调用栈信息、虚拟栈信息。

线程上下文——GetThreadContext

被监视进程的符号表——SymInitialize加载。

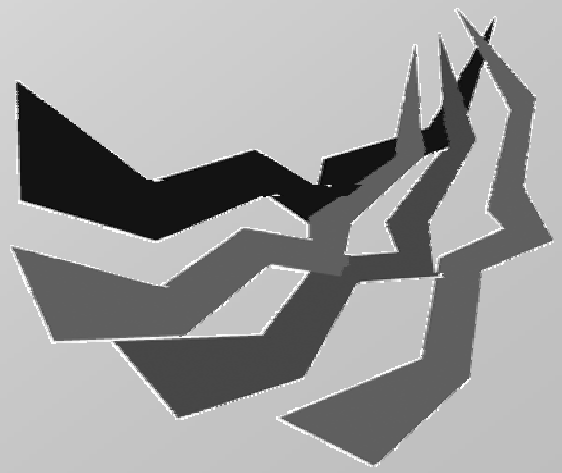
遍历被监测进程的堆栈——StackWalk遍历。

程序加载的模块信息——EnumerateLoadedModules

虚拟栈信息——保存，便于定位出错。

- 总结
  - 试图提出一个针对缓冲区溢出的发掘模型，但还有很多工作没有做。包括Fuzzer的自动化教本引擎等工作。同时，本文所叙述的工作中，仍有许多并没有进行效率的测试和在实践中进行检测。这都需要在将来进行完善和测试。

Thanks !



X'CON 2003