溢出植入型木马（后门）的原型实现

作者：FLASHSKY
邮箱：flashsky@xfocus.org
站点：www.xfocus.net

Xfocus 焦点峰会 2002

感谢与申明

- 感谢此文成型中获得的如下人员的讨论与支持
  - XUNDI，大鹰，冰河，ALERT7, BENJURRY,ISNO,REFDOM给予的技术讨论和提出了有益的意见
- 感谢所有到会成员给予的指点

申明：

作者无意实现一个木马，只是提供一种思路：将缓冲区溢出攻击和木马/后门相结合的木马实现手段，通过一个简单的原型来验证，并展示给大家这种实现方式的一些特点。作者提供关键代码段的技术实现的文档和演示来验证其实现，但不提供源代码和二进制程序，任何人都可以利用此文进行自己的技术研究和代码实现，但是自己负担自己开发程序进行非法行为的法律责任。

基本思路

- 木马（后门）如何有效的隐蔽?
- 溢出植入型木马（后门）的思路
- 溢出植入型木马（后门）的优势

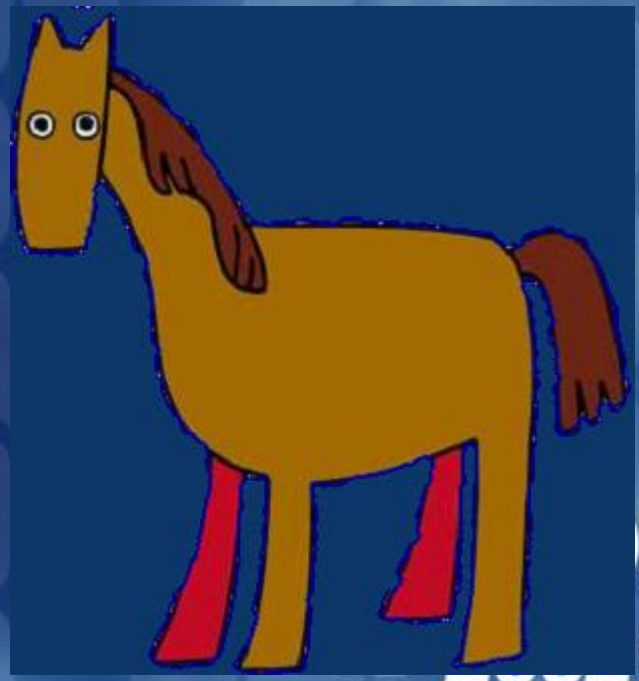
木马（后门）如何有效的隐蔽？

- 应用/代码本身的隐蔽
- 应用进程执行的隐蔽
- 自动启动相关的隐蔽
- 通讯的隐蔽

- 当前木马/后门发展的趋势
  - 驱动和内核级
  - 拦截系统调用服务来实现系统级的隐蔽
- 存在的问题
  - 代码量多
  - 影响一定的性能
  - 实现需要高的技巧与技能

溢出植入型木马（后门）的思路

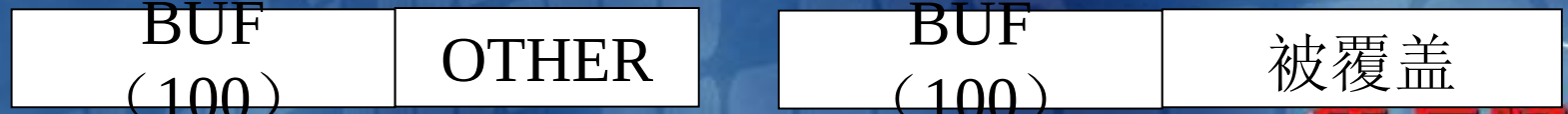
- 被动工作式的木马
- 瘦服务器端的木马
- 依赖于系统机制进行加载，和执行



溢出植入型木马（后门）的优势

- 为什么会选择溢出植入？
  - 溢出漏洞具备通用性
  - 溢出漏洞具备非常好的隐蔽性
  - 溢出本身就可以实现远程的控制
  - 溢出的攻击程序指令是由数据传输的，灵活，不留痕迹
  - 在服务内部进行，很容易实现进程，通讯，启动代码的隐藏
  - 完全被动方式工作，对性能等影响小。
  - 制造一个溢出漏洞比较简单和容易实现。

Recv(s,buf,100,0); ⟶ **Recv(s,buf,200,0);**



通用溢出漏洞的植入

- 通用化溢出漏洞要解决的4个问题
- 实现植入通用化溢出漏洞的思路
- 植入通用化溢出漏洞的实现
- 木马（后门）的在W2K下的实现

通用化溢出漏洞要解决的4个问题

- 溢出点定位
- JMP ESP代码提供和定位
- 溢出覆盖后对变量的引用访问违例
- 溢出覆盖后执行代码对溢出区的修改

实现植入通用化溢出漏洞的思路

- 扩展堆栈
- 植 入某个特定函数的替换转发函数

- 扩展堆栈

```
PUSH  EBP
MOV EBP，ESP
SUB ESP，XXX
（代码执行区）
ADD   ESP，XXX
PUSH  EBP
```

YYY>XXX

```
PUSH  EBP
MOV EBP，ESP
SUB ESP，YYY
（代码执行区）
ADD   ESP，YYY
PUSH  EBP
```

多出空间的作用

- 自动调整大小，使得溢出点一致
- 多余空间内放置SHELLCODE，不会被后面修改
- 可精心选择覆盖内容。避免访问违例

需要的条件

- 对传入参数引用使用EBP+XXX，对变量引用使用ESP+XXX的方式

- 替换转发函数

0X4444　　Recv函数地址

```
PUSH FLAG
PUSH XXX
PUSH BUF
PUSH S
CALL [0X4444]
```

0X4444　　RecvAdd函数地址

0X4c88　　Recv函数地址

```
PUSH FLAG
PUSH XXX
PUSH BUF
PUSH S
CALL [0x4444]
```

RecvAdd代码：

```
PUSH FLAG
PUSH YYY
PUSH BUF1
PUSH S
CALL [0x4c88]
COPY(BUF,BUF1,XXX)
RET
```

替代转发函数的作用

- 可以制造一个完全通用的溢出漏洞
  - 溢出点可控
  - 不会引发访问违例
  - 在溢出控制前BUF内容不会被修改
- 可以解决一些更多的技术问题

植入的通用化溢出漏洞的实现

- JMP ESP提供和定位
- **RECV**函数的替换转发函数实现
- 更深层次的利用

```
DWORD WINAPI recvadd(SOCKET s,char FAR* buf,int len,int flags){
        int num;   char buf1[0x1190];
        if(len>0x1000)
                num = recv(s,buf,len,flags);
        else{
                num = recv(s,buf1,0x11a9,flags); //扩大到标准指定的溢出点上
                if(num>0) { //判断是否收到包
                        if(num<=len)  //判断是否溢出，没有则拷贝内存
                                memcpy(buf,buf1,num);
                        else { //提供JMP ESP的地址
                                num=-1;
                                _asm{     mov             eax,1010101H
                                          mov             [esp+11A4H],eax;
                                }}}}
        return num;}
```

更深层次的利用

- 检测溢出和溢出返回地址
- 其他需要利用的变量提供
- 环境保护和线程的安全返回
- 新的替换转发函数的汇编实现

木马（后门）的在W2K下的实现

- 植入代码结构
- PE文件节点分析和代码的附加
- 附加代码的自动计算和替换
- 调用函数分析和导入表的替换

植入代码结构

PE文件结构

代码附加过程

- 分析代码节接点空间
- 找到需要替换的函数地址
- 在进程空间查找对应地址的导入地址
- 替换函数调用地址
  - 存在jmp [recv] 的形式，替换为jmp [recvadd]
  - 不存在jmp [recv] 的形式，直接替换每个call [recv]
- 保留真实的recv的地址，计算jmp esp代码和recvadd的地址，替换到植入代码的对应偏移处
- 附加代码，并进行相应PE头修改,自我删除

通用远程溢出的SHELLCODE

- 函数定位处理
- SOCKET复用
- 环境变量引用和环境上下文保护
- 溢出线程的安全返回

SOCKET复用

- SOCKET复用的意义
- 基本思路 (针对阻塞式SOCKET)
  - 获得有效的SOCKET描述符
  - 判断关联的SOCKET描述符的线程 (2次连接)
  - 挂起SOCKET相关的线程
  - 接管SOCKET通讯
- 实现代码
- 非阻塞式SOCKET的复用

环境变量引用和环境上下文保护

- 引用的变量和作用
  - SOCKET描述符号
  - 转发函数参数占用堆栈的大小
  - 函数返回地址
- 环境上下文和积存器的保护

溢出线程的安全返回

- 安全返回的意义
- 需要考虑的问题
  - 保存溢出SHELLCODE执行前的积存器内容
  - 保存需要返回的地址值
  - 计算恢复后的ESP/EBP，好放入对应的地址值。
  - 在恢复积存器后，还需要使用积存器读取返回地址值并放入返回前的ESP和读取函数在溢出前提供的函数参数大小来计算正常的ESP，因此对于积存器的保护需要一点技巧。
- 安全返回的代码实现
- 演示植入溢出到TEST服务并远程控制

阐发

- 饶过WIN的系统文件完整性保护（SFP）
- 溢出植入的通用化测试
- 探讨:只修改内存影象避免文件完整性检查

饶过WIN的SFP

- 饶过SFP的意义
- SFP的基本实现机理
- 饶过的思考
- 饶过SFP的实验

溢出植入的通用化测试

- 溢出植入的过程是完全通用的
- 溢出植入的替代函数是只针对函数级通用的
- 溢出植入远程控制端是完全通用且可根据需要灵活编写和变化的
- DNS服务的RECV溢出植入演示
- SQL SERVER服务的WSARECV溢出植入演示
  - 通用的WSARECV的替换函数
  - 重载异步IO的SOCKET的内存置换的问题