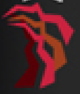
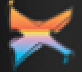
MSRPC Fuzzing with SPIKE 2006

Dave Aitel

www.immunityinc.com

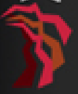
X'CON2006

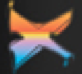




概览

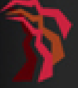
- Fuzzing 简介
- MSRPC快速简介和相关协议
- MSRPC fuzzing的历史和这些技术的缺点
- Immunity's专注于MSRPC fuzzing
- MSRPC的未来 (and hence, of MSRPC fuzzing)

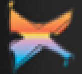




什么是Fuzzing?

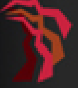
- 我们所有人都从把小字符串变成大字符串中获益
- Fuzzing所做的就是通过特殊的方法来对应用程序进行输入
  - 主要的好处：没有误报. 你通过fuzzing找到的所有bug都是可重现的（虽然它可能无法利用)
  - 主要的麻烦：速度很慢

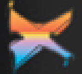




为什么要fuzz?

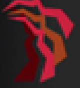
- 即便有安全公告或二进制比较工具，相对逆向工程而言，fuzzing更容易找到bug
- Fuzzing能找出通过二进制或源代码分析难以发现的bug
- A generalized fuzzer for a bug will tell you if a patch is good enough to cover edge cases or if it has an edge case that is still vulnerable

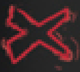




什么是 fuzz的最好方法?

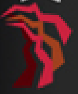
- Non-fault-injection approach
  - 我们不直接注射数据到程序的API中，因为它将导致误报
    - 例如绕过验证，对输入进行有效性检查等
- Fuzzing专注于发现可利用的漏洞
  - 这不是关于 QA – 只是关注我们感兴趣的,即整数溢出和缓冲区溢出

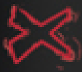
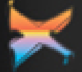




哪种类型的应用程序适合fuzzing

- 暴露在网络上的应用程序
  - 所有DCE-RPC!
- 闭源应用程序
- 看起来没有经过代码审核的应用程序
- 难于获取的应用程序
- 相对复杂的应用程序
  - 对复杂的应用程序进行审计将花费大量的金钱!

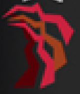


Fuzzing Mindset

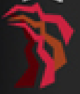
- There's a certain magic to a good fuzzer since there is no guarantee it will find anything
- Fuzzers将花费相当长的时间
  - 例如几周，几个月
- 当你开始开发一个fuzzer时，你要相信自己不是在浪费时间
- 大家看不起 fuzzer

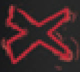
X'con 2006



Fuzzers的问题

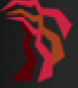
- Tokenization is rarely perfect
- 容易错失私有扩展
- Problem itself is exponential
- 一般只有在找到一些潜在的bugs后才能吸引blackhat
- But does give you a good initial indication of the “stance” of the application

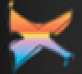




怎么打造 fuzzer

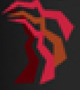
- Tokenization
- 产生正常的流量
- 产生异常的流量
- 检测和分析问题
- 分析 fuzzer的质量





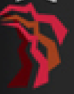
Tokenization

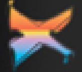
- 分解网络协议为常量(not fuzzed)和变量(fuzzed)
- 类型
  - 字符串，整数，大小，二进制块
- 通常常量都是头字符串，协议常量，网络交互的响应等
- Over-tokenization 将使你的fuzzer变得很慢
- Under-tokenization 将使你的 fuzzer发现不了任何东西



产生正常流量

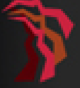
- 阅读和分解 RFC's 或其它可读的协议描述
  - fuzzing协议中没有实现的部分通常只会浪费你的时间
  - 将错失私有的扩展
- 对协议进行逆向工程
  - 能够半自动完成
  - If tool is flexible enough, human input can be invaluable
- 嗅探和静态分析
  - Even very dumb replay-and-bit-flipping can find many bugs
- 如果完成的不好，目标应用程序将忽略你的大部分流量





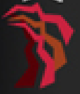
产生畸形

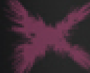
- Transforming normal traffic into malformed traffic, but in a way that is likely to cause exploitable problems
- Bit-flipping is most simplistic
  - For each bit we send, iterate over sending the opposite
- 改变流量中的一个部分将可能需要改变其它所有部分
  - 例如，内容长度检查

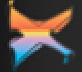


Fuzzing is not Fault Injection

- Fuzzing时你将从网络层开始测试所有层
- 在fault injection时，你直接向API插入错误的数据
- 有许多层你是不知道的
- Fuzzing 永不产生误报
- fault injection的缺点
  - 需要一个调试器，它将改变程序的操作
  - 产生误报

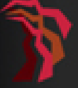


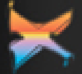




SPIKE简介

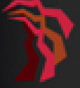
- 从2000年开始发布，最早的通用网络协议fuzzers之一
  - Greg的Hailstorm是另外一个类似的 (注意: 它与当前的 Hailstorm差别很大 – 之前的是一个商业的fuzzer，可用于任意协议)
- 介绍唯一的基于块(block-based)的fuzzing
- 包括HTTP,FTP及其它协议模块
- 用底层 C编写 (考虑速度)
- 基于 GNU 许可证来发布

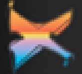




Block-based fuzzing

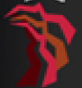
- 协议基本上都由同种元素组成
- 常量，块，变量
  - <invarient><size><varient><size><varient 2>
- 把每个变量都用测试的字符串替换，并更新相关的sizes
- 并且在每个变量的前面和后面插入测试字符串

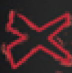
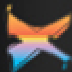




基于块fuzzing的优势

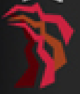
- 产生畸形流量的捷径
- Gut-feel：找到有趣 bugs
- Fuzz-streams 是可复用的
- 接近原始的有效流
- 能够比较容易的对被其它协议封装的协议进行fuzz

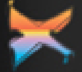


其它基于块的fuzzers

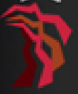
- Peach (基于Python的fuzzer)
  - 免费的
- Gleg.net ProtoVer (同样是基于Python)
  - 商业的

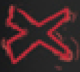




Immunity与MSRPC

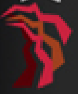
- 2000 – SPIKE, dcedump, ifids
- 2002 – CANVAS msrpc.py (支持验证、本地管道、SMB/DCE分片)
- 2003 – MSRPC 审计类
- 2004 – MOSDEF集成lexx.py和yacc.py
- 2005 – unmidl.py, DCEMarshall in CANVAS
- 2006 – SPIKE 2006

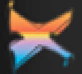




SPIKE 2006

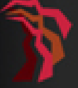
- 用Python重写并成为CANVAS攻击框架的一部分
  - 从包括DCE-RPC协议组的纯python网络协议库里获益
  - 更容易扩展和使用
- 增加基于字符串和整数的概念
  - A few selected fuzz-variables are used for EVERY variable in protocol while fuzzing
  - 找更隐藏的bugs
  - 程序变慢了 (但计算机速度在提高 : >)

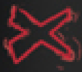
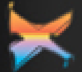




SPIKE选择的字符串

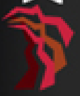
- 我们用字符 B来代替A，因为当我们发现一个堆溢出问题时，Windows内存管理标志中的B往往会让程序正常的崩溃
- 我们在所有长字符串前面都加上 \\、\\\\ 和 http://
- 我们使用长度从0到2200的字符串来捕捉单字节溢出问题
- 我们也会使用已知的常引发问题的特殊字符串集

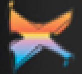


为何对MSRPC应用程序进行fuzz?

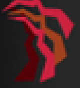
- 在微软默认的应用程序中有上千个可用的MSRPC接口
  - Writing Microsoft Windows exploits isn't going out of style any time soon
- 有很多厂商的MSRPC平台上进行开发
  - 这些厂商需要快速和简单的测试他们自己的接口
- Samba needs regression testing





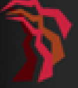
MSRPC 概览

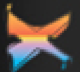
- 最初为 DCE-RPC, OncRPC和Corba的竞争者
  - 共享他们的安全问题
- Windows的DCOM中大量使用
  - 有很多扩展的使用
- 在商用unix中同样可用
  - SPIKE初始版本曾在AIX的实现中找到过bug
- 在Samba中实现了



MSRPC组成

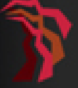
- 协议无关性
  - UDP/TCP/HTTP/NETBIOS/SMB/etc
- 数据类型无关性
  - Marshalling and demarshalling allows for encoding of complex data types (with pointers) as network streams
- 加密和验证
  - NTLM, security callbacks, etc
- 端点映射

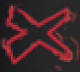




MSRPC 私有的

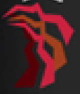
- 接口
  - UUID
- 接口版本
  - 主版本和次版本 (例如“1.0”)
- 函数编号
  - 大约从0到100





Free (as in speech) MSRPC tools

- Dcedump (port 135)
  - 获取可用的端点和接口列表
- Ifids
  - 获取特殊端点的接口列表
- Unmidl.py
  - 从可执行文件或DLL中产生IDL文件

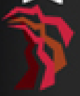


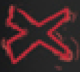
什么是 IDL?

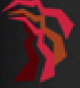
- “接口描述语言”
  - 解释使用了哪种数据类型，哪些函数可用及函数使用了哪些参数
- 通常厂商不会把IDL文件公开
  - 这使得产生有效流量变得困难
- 使用 “Microsoft IDL” 工具进行编译 (midl)

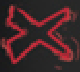
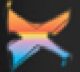
Unmidl tools

- 原始版本名为“muddle”，作者未知，基于GPL发布
- 基于Python的unmidl.py修复了复杂结构、指针等问题，基于GPL发布
- Followed by <3com product>
- Followed by <free product>



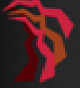




IDL示例（umpnp)

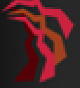
```
  long  Function_36( [in]  [string] wchar_t *
element_288,
    [in]  long  element_289,
    [size_is(element_291)] [in]  char
element_290,
    [in]  long  element_291,
    [size_is(element_293)] [out]  char
element_292,
    [in]  long  element_293,
    [in]  long  element_294
    );
```

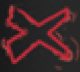


示例 2

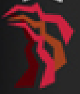
```
long  Function_09( [in]  [string] wchar_t *
element_825,
[in] [unique]  [string] wchar_t * element_826,
[in]  [string] wchar_t *  element_827,
[in]  [string] wchar_t *  element_828,
[in]  [string] wchar_t *  element_829,
[in] [unique]  [string] wchar_t * element_830,
[in] [unique]  [string] wchar_t * element_831,
[in] [unique]  [string] wchar_t * element_832,
[in] [unique]  TYPE_6 ** element_833,
[in] [unique]  TYPE_6 ** element_834,
[in]  long  element_835,
[out] [context_handle]  void * element_836
 );

 typedef  struct {
 [size_is(524)] char *element_774;
} TYPE_6;
```

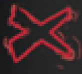
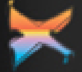
MSRPC Fuzzers的简要历史





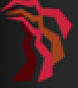
- SPIKE
- Samba SMBTorture
- Others?
  - LSD-PL MSRPC fuzzer + unmidl tool lead to MS03-026?
- SPIKE 2006!

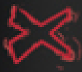
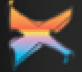


Interlude: VERDE

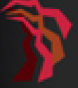
- 由SPIKE的早期版本发现
- 在XXXX服务中的任意内存free漏洞
- Immunity的Nicolas Waisman编写出可靠的利用代码
- 在 Windows 2000 SP4中被修复
- (简单演示)

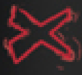
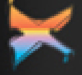


MSRPC fuzzing的难题

- 很难创建有效的协议流
  - Windows 2000 and above check for rigorous protocol compliance – IDL 文件必须正确!
  - IDL files are not a one-to-one match with demarshalling
- 必须包括验证
- 上下文处理
- 接口也许只能从本地访问
  - CANVAS 支持本地命名管道

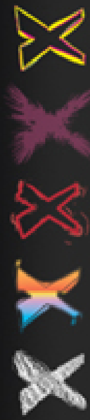


SPIKE 2006与之前版本的不同之处

- 改进unmidl.py
- Working dcemarshaller
  - 能够fuzz复杂的指针结构和类型
- SPIKE 为size_of() 参数提供解决方案
- SPIKE 2006能够fuzz几乎所有类型的端点(包括HTTP, local等)
- Response_trigger寻找信息泄露，异常响应

Fuzzing流程



Create IDL using unmidl.py

Convert each function to marshaller (automate this!)

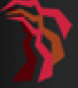
Examine need for context_handles or auth

Use dcedump/ifids to find endpoint (TCP/named pipes done automatically)

dcefuzz!

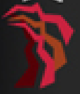
Create get_context_handle()

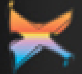
Create response_trigger()



Fuzzing Metrics

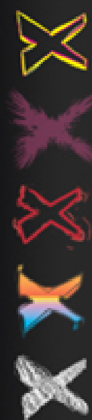
- Measuring fuzzers in “number of tests” is like measuring computers in kilograms
- 代码覆盖不是程序状态覆盖
  - 如果函数A只在调用函数B后崩溃，之后你覆盖到这两个函数，发现仍旧不能使它崩溃
    - 这比你想像的要更加常见
  - 必须用正确的输入来覆盖代码，从而发现bugs
- Concurrency bugs are hard to “measure”
- 每个fuzzer找到的bug都不一样





Fuzzing Metrics (cont)

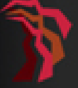
- 我们现在能够做的:
  - 新的fuzzer能找出所有的已知bug（自动的）和发现一些新的bug吗?
  - 对协议进行fuzz比逆向工程更快更容易吗？
  - Fuzzer能在合理的时间内完成任务吗？

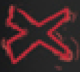


SPIKE 2006成果

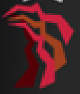
- 平均小于一小时完成对一个指定函数的fuzz
- 发现先前已知的漏洞
- 演示
  - umpnp
  - Exchange DoS
  - …

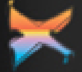
SPIKE 2006和MSRPC Fuzzing的未来





- Automatic fuzzer creation from unmidl and unmidl improvements
- VisualFuzz – Apply SPIKE 2006 techniques via a visual language (like Immunity VisualSploit)
- Use Immunity Debugger
  - To analyze coverage of MSRPC functions
    - 不是覆盖整个DLL，而是覆盖MSRPC函数入口点下面所有潜在代码
  - 创建更多正确的IDL文件





结论

- Fuzzing MSRPC 带来一些有趣的问题,但它们大部分都被SPIKE数据结构解决了
- Block-based fuzzing scales up to complex protocols
- Questions?