# Advanced Windows Exploitation

Dave Aitel
Immunity, Inc
http://www.immunitysec.com/

# Agenda

- What is Immunity?
- Windows for Unix Hackers
- DCE-RPC
- Finding bugs with SPIKE
- MS-SQL
- The shellcode problem
- Heap Overflows
- IIS
- Demos, other fun

# Immunity, Inc

- New York City based Corporation
- 7 Months old, privately financed
- Information Security Services
  - Application focus
  - Protocol Analysis
  - Training
  - Cutting Edge Products
    - CANVAS
    - BODYGUARD
    - SPIKE, SPIKE Proxy

# Windows for Unix Hackers

- Windows
- X86
- Component Architecture
- Privilege tokens
- Threaded
- Closed Source

- Unix
- x86/RISC
- Process architecture
- User ID
- Forked
- Open Source

# X86

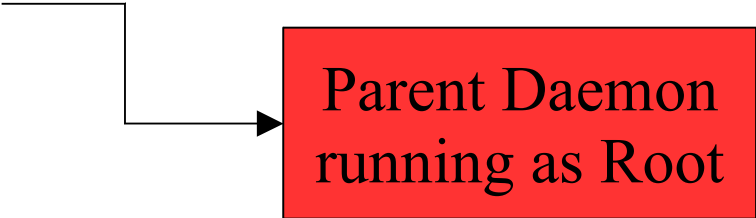
- Unaligned address references
  - Except ESP,EBP which must be word aligned for internal Windows API calls to work properly
- No instruction cache (post 486), register windows, or other painful RISC idioms

# Windows' Component Architecture

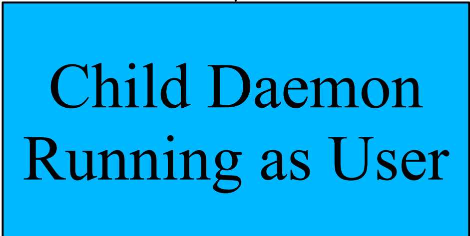
- No setuid programs, all privilege comes directly from the kernel
- lsass.exe (local security authority process)
- DCE-RPC
- Impersonation

# The Unix Way

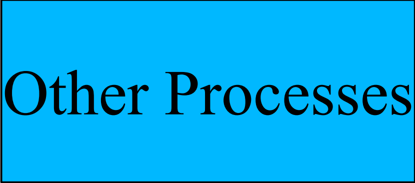
Client
Connection



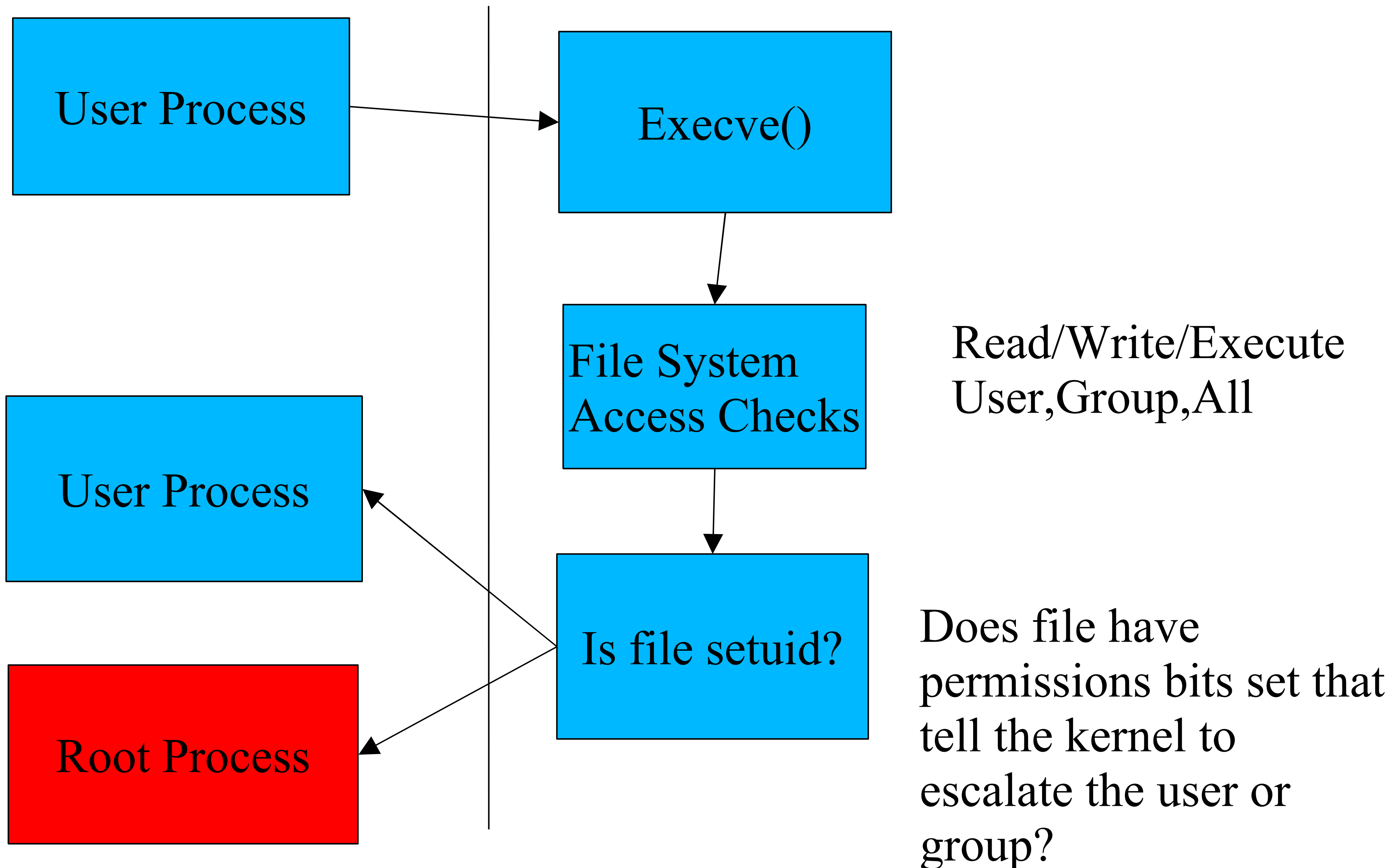

Fork(); setuid()



File System



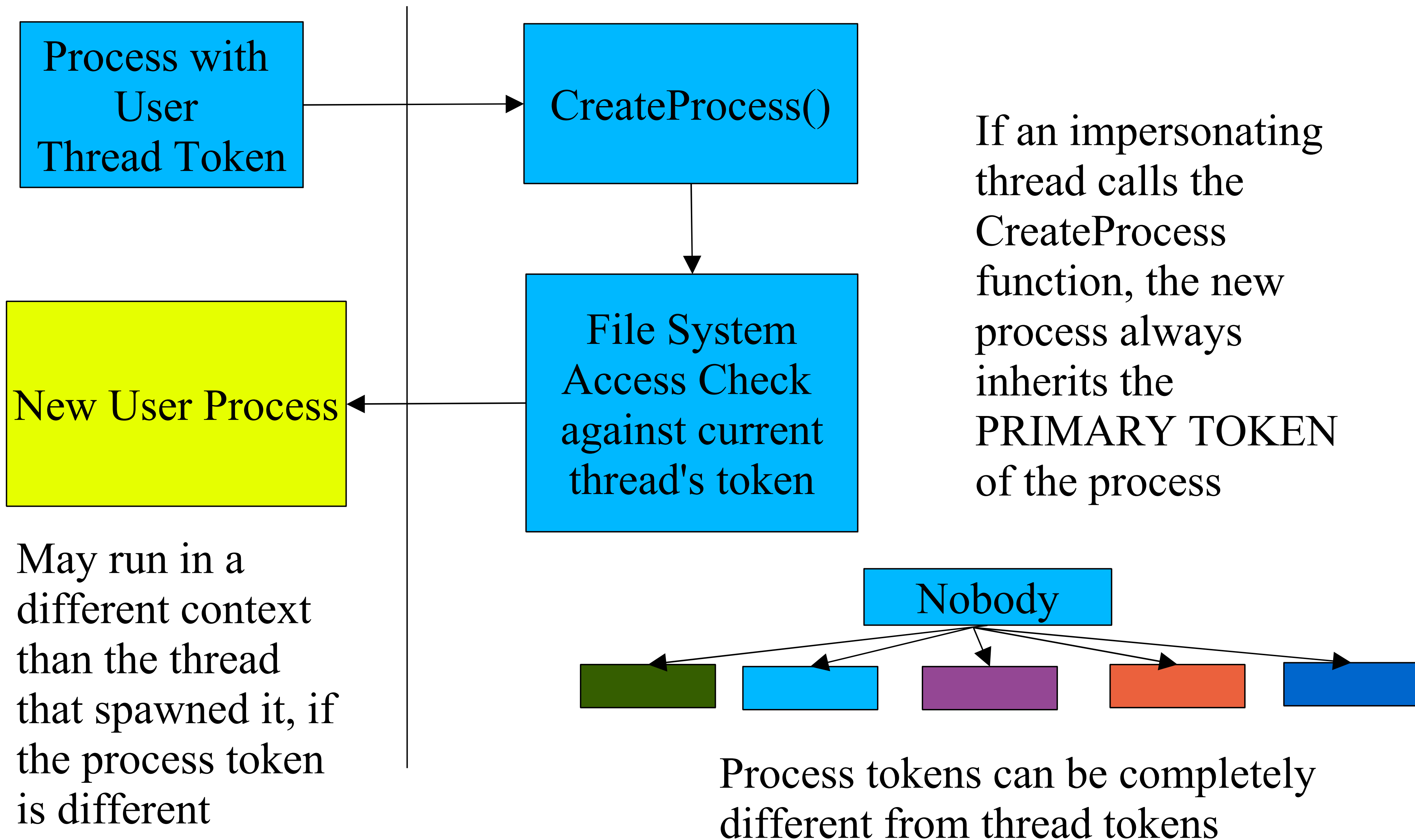
# Spawning Processes Under Unix



# What comes along for the ride?

- All open file handles
  - Includes special purpose device files like /dev/kmem, /dev/mem, raw sockets etc
  - Except those specifically set close-on-exec
- User ID, Group Ids
- Environment

# Windows Process Spawning



# What gets carried over?

- The current “Desktop” (Shatter!)
- Current Working Directory
- Specified Environment
- Any “handles” set to be inherited (and explicitly passed!)
- A console, if it's a console application
- Standard input and output

# Handles can be:

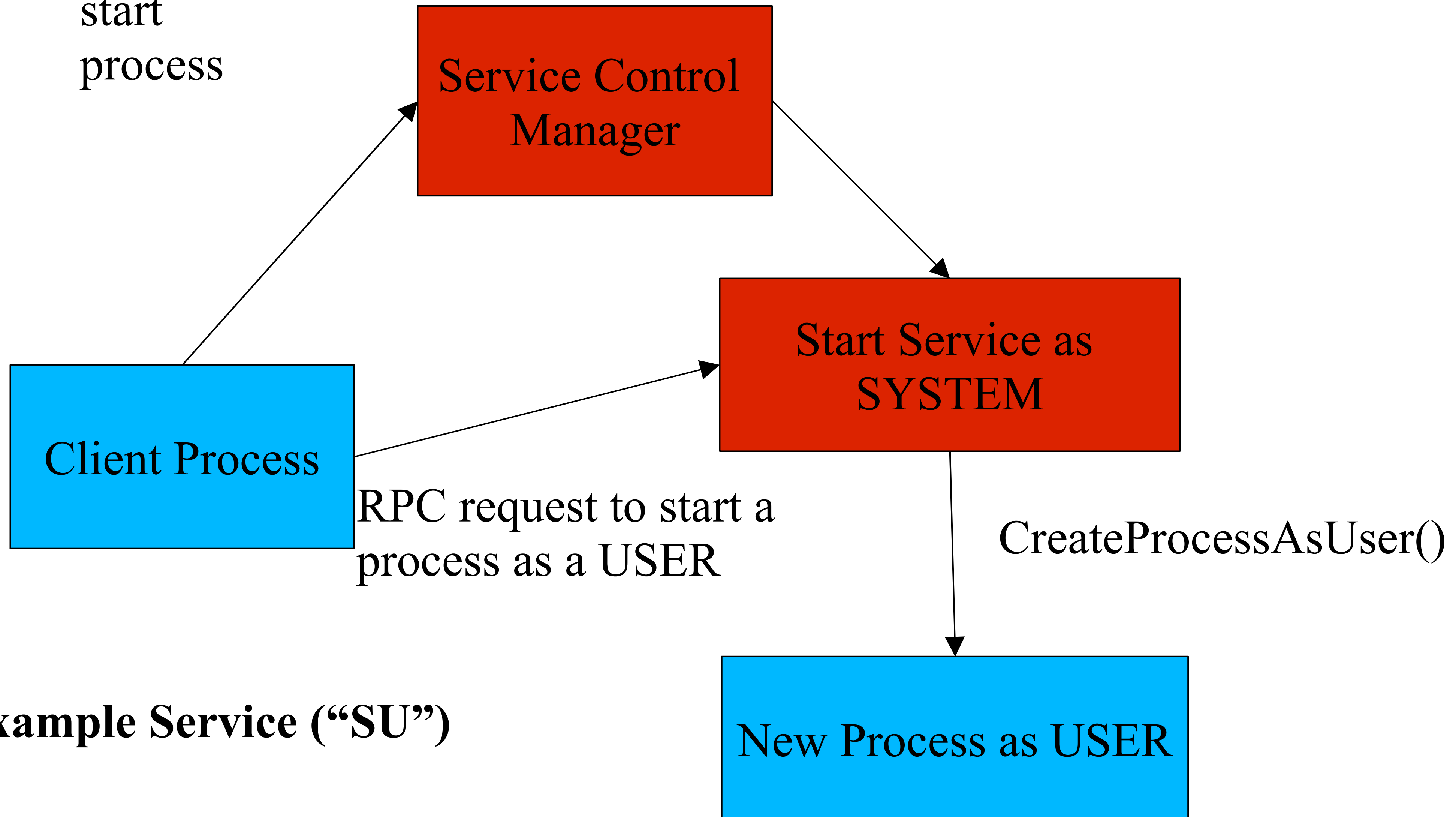
- Open Files
- Processes
- Threads
- Mutexes
- Events
- Semaphores
- Pipes
- File Mapping object
- Buffers
- Mailslots

Child processes can not inherit memory handles, DLL Module handles, GDI handles, or USER handles

Psuedohandles are also not inheritable (such as those returned by GetCurrentThread())

Child processes must be explicitly passed inherited handles, via IO or some other inter-process communication method (RPC, for example). This is done with the function call "DuplicateHandle()"

# Services in Win32



# Summary

- Windows by default has less exposure to inherited resources than Unix
- Nobody understands RPC, so finding local exploits can be difficult
- You can run a process which cannot read it's own exe file! (Exploiting IIS dllhost.exe does this)

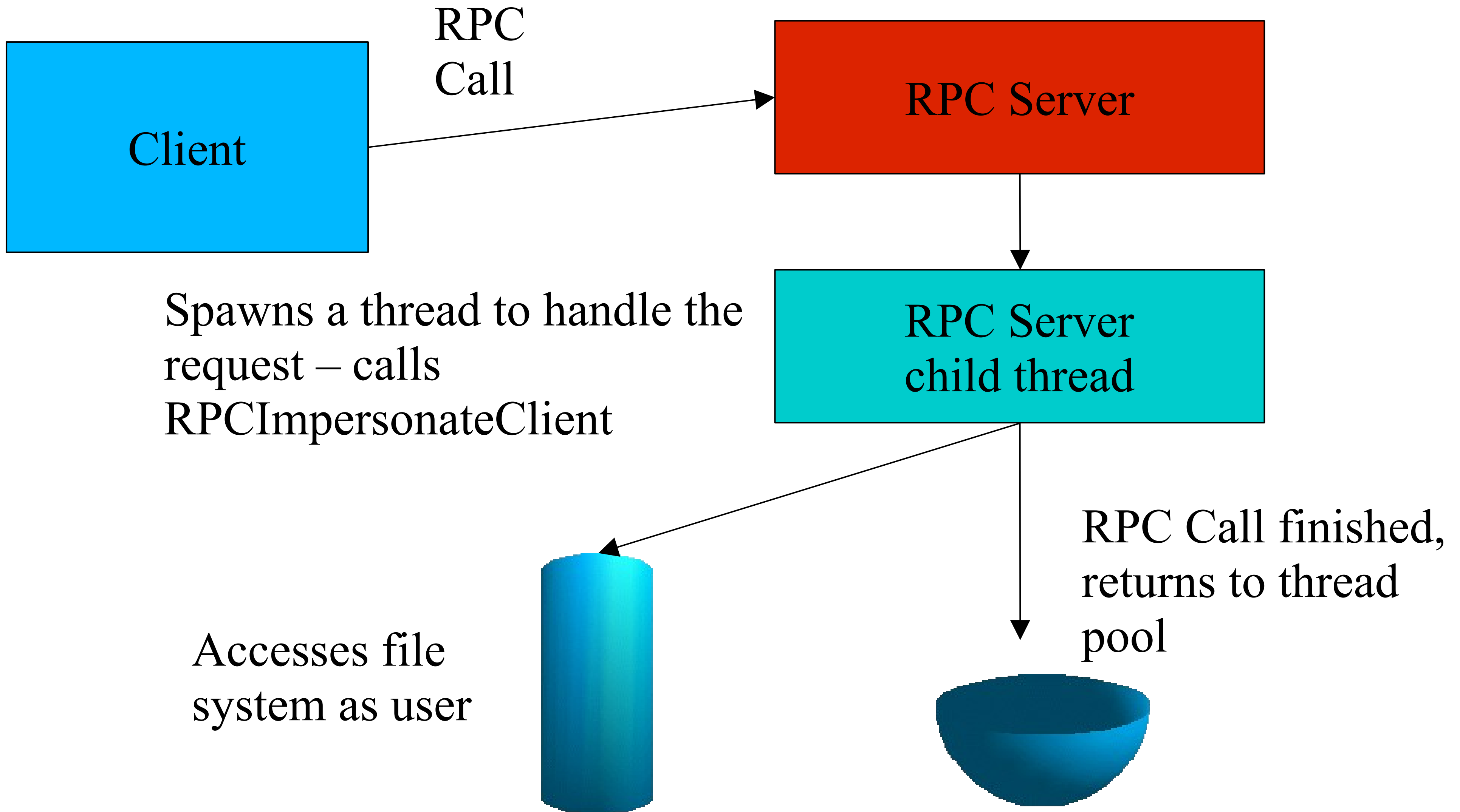
# What's a Token

- Impersonation and Access under Windows is infinitely complex
  - ACE, DACLs, Privileges, UID, GIDs, Cloaking, etc
- A token is like a smart card that a THREAD (not a process) can carry with it and present to the kernel whenever access is checked
  - What user I am
  - What I can do as that user
- Flexibility++==Security--

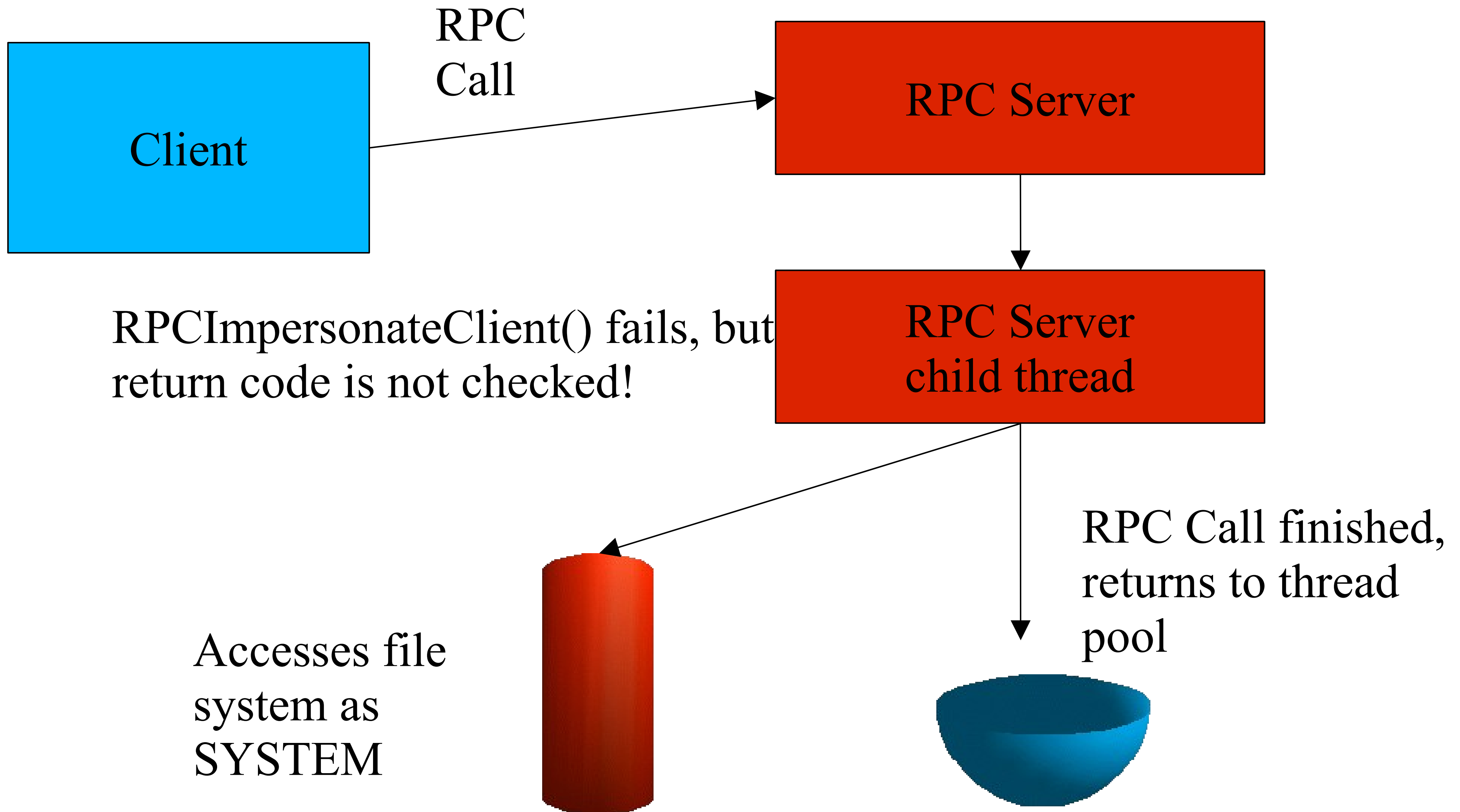
# The Token Stork

- Where do tokens come from?
  - lsass.exe
    - Or any process with similar privileges
    - LogonUser() + CreateProcessAsUser
  - Impersonation
    - Any client on your named pipe
    - Any connection to your RPC service
    - ImpersonateDDEClientWindow(), ImpersonateNamedPipeClient(), RPCImpersonateClient()

# RPC Impersonation



# When RPC Impersonation Fails



# To Sum Up

- NT uses RPC in place of setuid files
  - Services are not vulnerable to environment variable and argument overflows the way setuid programs are
  - RPC arguments are fair game though
    - These are not well documented
- NT uses thread tokens instead of fork+setuid()
  - Tokens are per-thread, not per-process
  - CreateProcess() doesn't carry as many resources with it as Unix, but it's not used for typical daemon services

# How does all this gibberish about tokens affect my overflows?

- All those DCE-RPC services are available remotely via TCP and/or UDP!
- Undocumented DCE-RPC services are behind everything, doing the real work. When you find an overflow, you may be in a completely different process than the server itself
- Lack of per-thread memory protection is an exploitation goldmine
- Focus on multi-threaded processes makes the stack completely unreliable
  - The heap is unreliable too, the only thing you can rely on is where a process's text (code) pages are, and even that is dependent on the program version

# Finding DCE-RPC Bugs

- SPIKE
  - (http://www.immunitysec.com/spike.html)
  - Implements a DCE-RPC stack, with a built in fuzzer
- IDA-Pro
  - Look for those NDR_ functions
- IDL files (Interface Description Language)
  - Assuming you have them
- Ethereal dissectors

# DCE-RPC Bugs Found With SPIKE

- Exchange 2000 “DoS”, function 0
- Exchange 2000 “DoS”, function 5
- 1 DoS on SVCHOST.EXE (port 135 TCP)
  - Windows 2000-XP (NT not tested)
- mstask.exe

# Like SunRPC, DCE-RPC has:

- SunRPC Program Number (100000)
- Portmapper (port 111)
- Function numbers
- Program Versions

- UUID Service Number
  - One process can service many functions, also like SunRPC
- Portmapper (port 135)
- Function numbers
- Program versions

# So to directly fuzz DCE-RPC

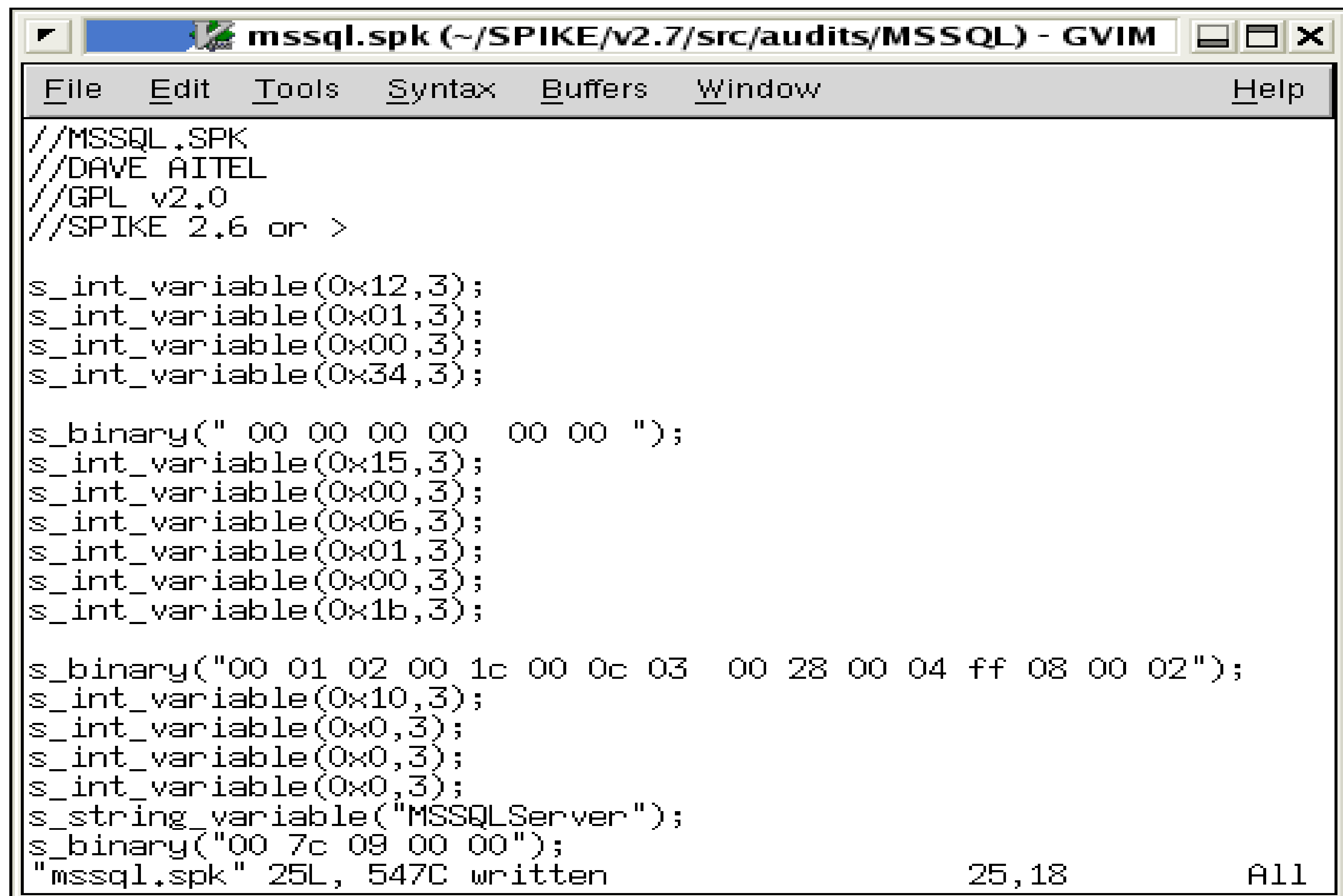
- ./msrpcfuzz target port SERVICEUUID Version VersionMinor FunctionNumber NumberofTries NumberofItems
- ./Dcedump target to get ports running tcp services
- ./Ifids target port to get all the services running on a port
- Attach with Ollydbg, write the exploits!
- Everything is Free, GPLed.

# What about one of the many closed protocols in, say, MS-SQL?

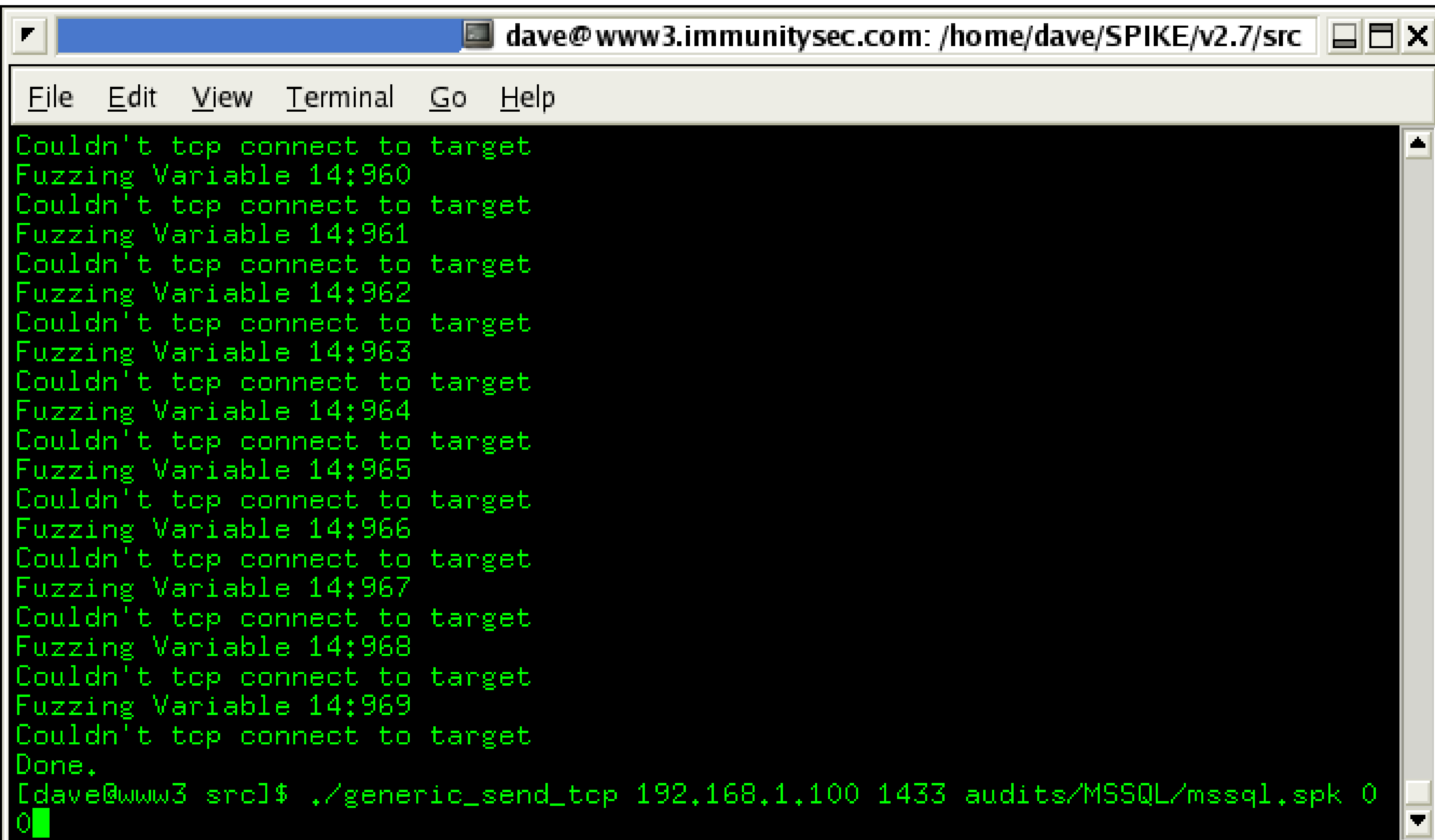
- To fuzz a closed source protocol with SPIKE, first find a client of some kind
  - ISQLW.exe, in this case
- Connect with the client and store off the network traffic with Ethereal
- Massage the data into a SPIKE script
- Run the SPIKE script against MS-SQL, see if it crashes
- Write the exploit

# Capture Some Traffic and Convert it to a SPIKE Script

```
00000000  12 01 00 34 00 00 00 00  00 00 15 00 06 01 00 1b ...4.... ........
00000010  00 01 02 00 1c 00 0c 03  00 28 00 04 ff 08 00 02 ........ .(..ÿ...
00000020  10 00 00 00 4d 53 53 51  4c 53 65 72 76 65 72 00 ....MSSQ LServer.
00000030  7c 09 00 00                                      |...
```



# How to run the SPIKE Script



OllyDbg - sqlservr.exe

File View Debug Plugins Options Window Help

L E M T W C K B R ... S

CPU - thread 00000938, module SSNETLIB

```
42CF72B4   8A02            MOV AL,BYTE PTR DS:[EDX]
42CF72B6   8945 EC         MOV DWORD PTR SS:[EBP-14],EAX
42CF72B9   8B4D 0C         MOV ECX,DWORD PTR SS:[EBP+C]
42CF72BC   034D FC         ADD ECX,DWORD PTR SS:[EBP-4]
42CF72BF   33D2            XOR EDX,EDX
42CF72C1   8A51 01         MOV DL,BYTE PTR DS:[ECX+1]
42CF72C4   C1E2 08         SHL EDX,8
42CF72C7   8B45 0C         MOV EAX,DWORD PTR SS:[EBP+C]
42CF72CA   0345 FC         ADD EAX,DWORD PTR SS:[EBP-4]
42CF72CD   33C9            XOR ECX,ECX
42CF72CF   8A48 02         MOV CL,BYTE PTR DS:[EAX+2]
42CF72D2   03D1            ADD EDX,ECX
42CF72D4   8995 E8FDFFFF   MOV DWORD PTR SS:[EBP-218],EDX
42CF72DA   8B55 EC         MOV EDX,DWORD PTR SS:[EBP-14]
42CF72DD   8995 D0FDFFFF   MOV DWORD PTR SS:[EBP-230],EDX
42CF72E3   83BD D0FDFFFF 0 CMP DWORD PTR SS:[EBP-230],1
42CF72EA  v0F84 92000000   JE SSNETLIB.42CF7382
42CF72F0   83BD D0FDFFFF 0 CMP DWORD PTR SS:[EBP-230],2
42CF72F7  v74 0E           JE SHORT SSNETLIB.42CF7307
42CF72F9   83BD D0FDFFFF 0 CMP DWORD PTR SS:[EBP-230],3
42CF7300  v74 5E           JE SHORT SSNETLIB.42CF7360
42CF7302  vE9 1B010000     JMP SSNETLIB.42CF7422
42CF7307   8B45 0C         MOV EAX,DWORD PTR SS:[EBP+C]
42CF730A   0385 E8FDFFFF   ADD EAX,DWORD PTR SS:[EBP-218]
42CF7310   50              PUSH EAX
42CF7311   8D8D ECFDFFFF   LEA ECX,DWORD PTR SS:[EBP-214]
42CF7317   51              PUSH ECX
42CF7318   E8 A5700000     CALL <JMP.&MSVCRT.strcpy>
42CF731D   83C4 08         ADD ESP,8
42CF7320   68 A01ED042     PUSH SSNETLIB.42D01EA0
42CF7325   8D95 ECFDFFFF   LEA EDX,DWORD PTR SS:[EBP-214]
42CF732B   52              PUSH EDX
42CF732C   E8 97700000     CALL <JMP.&MSVCRT.strcmp>
42CF7331   83C4 08         ADD ESP,8
42CF7334   85C0            TEST EAX,EAX
42CF7336  v74 1F           JE SHORT SSNETLIB.42CF7357
42CF7338   68 002CD042     PUSH SSNETLIB.42D02C00          ASCII "MSSQLSer
42CF733D   8D85 ECFDFFFF   LEA EAX,DWORD PTR SS:[EBP-214]
42CF7343   50              PUSH EAX
42CF7344   FF15 D8F0CF42   CALL DWORD PTR DS:[<&MSVCRT._stricmp>]  MSVCRT._stricmp
42CF734A   83C4 08         ADD ESP,8
42CF734D   85C0            TEST EAX,EAX
42CF734F  v74 06           JE SHORT SSNETLIB.42CF7357
42CF7351   C645 F7 01      MOV BYTE PTR SS:[EBP-9],1
42CF7355  vEB 04           JMP SHORT SSNETLIB.42CF735B
42CF7357   C645 F7 00      MOV BYTE PTR SS:[EBP-9],0
42CF735B  vE9 C2000000     JMP SSNETLIB.42CF7422
42CF7360   8B4D 0C         MOV ECX,DWORD PTR SS:[EBP+C]
42CF7363   038D E8FDFFFF   ADD ECX,DWORD PTR SS:[EBP-218]
```

Registers (MMX)

```
EAX 00000000
ECX 00000061
EDX 82828287
EBX 192BA090
ESP 2A9BE724
EBP 2A9BE958 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ESI 00000040
EDI 2A9BE9CC ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

EIP 42CF72B4 SSNETLIB.42CF72B4

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FF98000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

MM0 0000 0000 0000 0000
MM1 0000 0000 0000 0000
MM2 0000 0000 0000 0000
MM3 0000 0000 0000 0000
MM4 0000 0000 0000 0000
MM5 0000 0000 0000 0000
MM6 0000 0000 0000 0000
MM7 BE75 7844 B53B B800
```

Call stack of thread 00000938

| Address | Stack | Procedure / arguments | Called from |
|---|---|---|---|

```
Address  Value    ASCII Comment
00A03000 28EC1BC0
00A03004 28E2C030
00A03008 00000080 Ç...
00A0300C 00000080 Ç...
00A03010 190E00B8
00A03014 28E300B8
00A03018 28E340B8
00A0301C 00000010
00A03020 00000000 ....
00A03024 00000000 ....
00A03028 00000064 d...
00A0302C 00007FFF
00A03030 00000040 @...
00A03034 0000000C ....
00A03038 00000000 ....
00A0303C 00000000 ....
00A03040 00000000 ....
00A03044 0000001E
```

```
2A9BE6FC 2A9BE9CC ASCII "AAA
2A9BE700 00000040
2A9BE704 192BA090
2A9BE708 2A9BE958 ASCII "AAA
2A9BE70C 78003096 MSVCRT.780
2A9BE710 2A9BE744 ASCII "AAA
2A9BE714 00000061
2A9BE718 42CF734A SSNETLIB.4
2A9BE71C 2A9BE744 ASCII "AAA
2A9BE720 42D02C00 ASCII "MSS
2A9BE724 00000003
2A9BE728 00000002
2A9BE72C 00000000
2A9BE730 00000004
2A9BE734 19306438 ASCII "¦o"
2A9BE738 2A9BE948
2A9BE73C 00000000
2A9BE740 0000001C
2A9BE744 41414141
```

Access violation when reading [82828287] - use Shift+F7/F8/F9 to pass exception to program    Paused

# The details of the MSSQL Hello Vulnerability

- Typical stack overflow
  - Redirect EIP to 0x42ae1ec9 or 0x42ae1eb9 (jmp edi)
  - Set some pointers to 0x751b8181 (a writable portion of memory) so the program does not cause an exception before it returns
  - Program recovers cleanly after exploitation
  - Executes your shellcode as LOCAL/SYSTEM on every system I've tried it against

# Shellcode (Unix Vs. Win32)

- System Calls are int 0x80 (or similar)
  - Easy to write small shellcode that calls out to a remote host or executes an arbitrary command
- Dynamic libraries and symbols are accessed via dlopen() and dlsym()
  - Very difficult to find, involves opening /proc/self/maps (see grugq's paper)

- “system calls” are interrupt driven, but take 500 arguments each, and change every OS revision
- Dynamic libraries and symbols are easy to find with loadlibrary() and getprocaddress()
  - But how do you find loadlibrary() and getprocaddress()?

# Finding LoadLibrary() and getprocaddress()

- Assume they are at a particular place in kernel32.dll, as mapped into the process
  - Per OS version
- Assume they are imported into a known place in a function table in the process (the Import table for example)
  - Per process version

# Parse memory intelligently to find GetProcAddress and LoadLibrary

- NSFOCUS (aspcode.c)
  - Set an exception handler so bad memory reads don't exit the shellcode
  - Start at 0x77e00000 and blindly hunt to find the Kernel32.dll page
  - Parse that to find GetProcAddress()
  - Call GetProcAddress() to find LoadLibraryA()

# Parsing PE Headers

- Greg Hoglund's (www.rootkit.com) Buffer Overflow Kit for Windows
  - Start at 0x0040003C
  - Find Import Lookup Table from that
  - Loop over DLL's and compare every function in the DLL against a hash of GetProcAddress and LoadLibraryA
  - Smallest code I've seen to do this

# Virus Writer's Lessons

- http://www.builder.cz/art/asembler/anti_procdump.html
  - fs:30h is pointer to PEB
    - This is always the case
  - *that + 0c is PEB_LDR_DATA pointer
  - *that +0c is load order module list pointer
- With a list of the module bases, you can go to each PE header, matching the names against KERNEL32.dll
- Inside Kernel32's Export Table are the pointers to the functions you want, and their names to match against
- Search on PECOFF at MSDN site to see detailed description of all of these structures

# A brief word on encoder/decoders

- Decoders are the tiny stubs of assembly language code that have to pass through arbitrary filters
- Decoders are typically the only parts of the shellcode that can trigger an IDS
  - Hence, many are kept secret
- Phrack Magazine's asc.c is a decoder creator that creates printable ASCII code for arbitrary shellcode at a 12-1 expansion
- Decoders in x86 for almost any filter exist, including Unicode strings, printable ascii strings, upper case, lowercase, or simple “no special characters” filters
- CANVAS includes a UNICODE and Additive encoder/decoder

## Why Additive and not XOR?

- An additive encoder/decoder simply executes KEY+A where A is every word in the encoded shellcode
- XOR cannot replace single bits – if the filter is disallow(BYTE & 0x01) then XOR can't possibly fit
- Disadvantages of Additive
  - Random guessing strategy for generating keys is much slower than XOR key generation
  - Still doesn't fit very restrictive filters

# What should shellcode do?

- Shellcode cannot maintain secrecy
  - Hacking in the clear is for amatures
  - RSA and Key Generation is hard in ASM
- Shellcode typically is operating inside a program as a parasite
  - You are holding things up
    - Detach from the program quickly so it can handle other people's requests.
  - You have special tokens and handles available to you in your memory space
  - You may be unstable
    - Heap, or other global variables may be trashed
    - Other requests may be messing things up

# Additional Win32 Weirdness

- ESP must be word aligned for some function calls to work properly
  - Socket() calls, especially
- You never know where the temp directory is
  - c:\winnt\temp?
  - d:\winnt\temp?
  - Sometimes the current directory is not writable (by your user token)
- Hence shellcode cannot have a hard coded place to write a file

# 580 Bytes: What my shellcode does

- Calls out to a remote server
- Executes arbitrary functions on behalf of that server
  - Finds a writable directory, downloads a file to that directory, and executes it
- Exits the current thread
- Future Projects:
  - Grabbing tokens and comparing them to Local/System or Admin!
  - Repairing heaps

# Demo of CANVAS MSSQL HELLO

- CANVAS is a commercial grade pure-Python Exploitation Toolkit
- http://www.immunitysec.com/CANVAS/

# Heap Overflows

- Unix heap overflows are exploitable by using a fake chunk to overwrite a function pointer
- That function pointer is in the Global Offset Table
  - Per OS version and program version

- Win32 heap overflows are exploitable by using a fake chunk to overwrite a function pointer
- That function pointer is the global exception handler
  - Per OS Version

# Advanced Heap Manipulation on Win32

- Manipulating heap structures properly allows you to write an instruction (jmp esp, for example) to memory somewhere, then overwrite global exception handler with that address as the target
- When the program next has an exception it will jmp esp!
- XP actually dereferences, so you can exploit it 100% of the time by finding a pointer to your buffer somewhere in memory that does not change
  - Try OLE's pointers, they always work for me

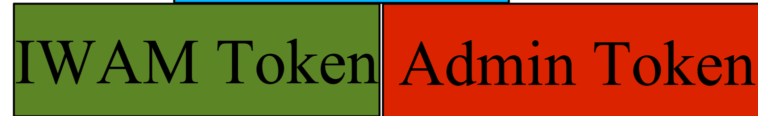
# Back to the basics

- Let's say that a double write is not possible, how does a heap overflow exploit typically work?
- When a heap overflow's exception occurs there is often no register pointing to the attacking string
  - Attacker fills up as much of the heap as possible with nops and shellcode
  - Attacker overwrites the global exception pointer to point into the heap
  - An exception occurs,
    - and the shellcode is run
    - Or the program crashes and gets restarted

# IIS

Heap Overflow in
HTR, ASP, MSADC

RPC




Improper cloaking
means dllhost can
impersonate system!

This Admin token is sometimes sitting around

Overflow occurs in
thread with IWAM
token

# IIS Token Weirdness

- Because IUSR is the primary process token, and IWAM is the current thread's token
  - Files are written as IWAM
  - CreateProcess() uses IUSR
- Spawned processes cannot execute or read their own .exe
- It should be possible to hunt down the SYSTEM token if it happens to be there, and use that instead!

# UTF-16 for Fun and Profit

- C char is often changed to wchar internally or specifically by a programmer in Win32
- Wchar can be up to 4 times the length, but most people only calculate for twice the length
  - Values above 0x7f are represented as 0xc200ac20 (for 0xff, as an example)

# Conclusion

- Understanding Windows's Security Model is essential for proper exploitation
- DCE-RPC framework is nightmarishly complex, which means it is full of holes
- Heap and stack overflow techniques are as advanced on the win32 platform as on Unix platforms
- Still many low hanging fruit in closed source applications waiting to be found
- Questions?

# Immunity

## Products



For More Information Contact: dave@immunitysec.com

Copyright 2002 - Immunity Security, Inc. (NYC)

ILLUSTRATE TRUE RISK

# CANVAS

- Price
  - $995 for initial purchase, comes with 3 months of free updates
  - Additional updates are $495 for 3 months
  - Enterprise Licenses Only
  - Full Source Code Included (Python)
- More information
  - http://www.immunitysec.com/CANVAS/

# The Problem

- IS Analysts rarely know the true nature of vulnerabilities
  - Does this vulnerability affect my systems?
  - What danger does this attack pose to my configuration?
  - How can I show management the true risks?
  - Does my IDS/Managed Security Service really detect this attack?

# CANVAS's Solution

- Polished and Profesional Exploit Toolkit
  - Completely Open Architecture
    - Scriptable, modifiable, customizable
  - Updated Constantly
  - Focused on Your Greatest Pain
    - IIS
    - MS-SQL
    - Coldfusion
  - Python codebase ensures portability to Windows, Unix, Mac, or anything else

# CANVAS Technology

- Service Pack independent Win32 Syscall-Redirection shellcode
- Encoder/Decoders for x86
  - Unicode
  - Additive
- Exploit development Python framework
  - String manipulation
  - Integer manipulation and unsigned integer emulation

# Completed CANVAS Vulnerability Modules

- IIS ASP Chunked Heap Overflow
- MS-SQL Server Hello Stack Overflow
- IIS MSADC Heap Overflow
- Each of these can be
  - demonstrated to upper management
  - scripted as an advanced vulnerability assessment tool
  - used to accurately test your IDS system
  - or otherwise used by your organization
- CANVAS vulnerabilities sometimes are released to CANVAS before checks are placed into Nessus or other vulnerability scanning mechanisms
- CANVAS modules allow you to recognize the after-affects of attack, unlike a vulnerability scanning program

# Immunity

## Products



For More Information Contact: dave@immunitysec.com

Copyright 2002 - Immunity Security, Inc. (NYC)

# Other Immunity Products

Finds kernel trojans on Solaris 2.6-2.8
US$20,000 for a enterprise license

Locates web application vulnerabilities. Includes spidering, scanning, form password brute forcing, and overflow checks. Pure Python. GPL.

Sophisticated C API for analyzing arbitrary network protocols. Includes several examples. GPL.