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

- Parent Daemon running as Root
  - Fork(); setuid()
  - Child Daemon Running as User

Other Processes

File System

Client Connection
Spawning Processes Under Unix

User Process -> Execve() -> File System Access Checks

- Is file setuid?
- Read/Write/Execute User, Group, All

Does file have permissions bits set that tell the kernel to escalate the user or group?
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

Process with User Thread Token

CreateProcess()

File System Access Check against current thread's token

If an impersonating thread calls the CreateProcess function, the new process always inherits the PRIMARY TOKEN of the process

New User Process

May run in a different context than the thread that spawned it, if the process token is different

Nobody

Process tokens can be completely different from thread tokens
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

RPC request to start process

Service Control Manager

Start Service as SYSTEM

RPC request to start a process as a USER

CreateProcessAsUser()

Example Service ("SU")

New Process as USER
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 its 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

Client

RPC Call

RPC Server

Spawns a thread to handle the request – calls RPCImpersonateClient

RPC Server child thread

Accesses file system as user

RPC Call finished, returns to thread pool
When RPC Impersonation Fails

RPC ImpersonateClient() fails, but return code is not checked!

Accesses file system as SYSTEM

RPC Call finished, returns to thread pool
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](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
- 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

```
//MSSQL.SPK
//DAVE AITEL
//GPL v2.0
//SPIKE 2.6 or >

s_int_variable(0x12,3);
s_int_variable(0x01,3);
s_int_variable(0x00,3);
s_int_variable(0x34,3);

s_binary(" 00 00 00 00 00 00 ");
s_int_variable(0x15,3);
s_int_variable(0x00,3);
s_int_variable(0x06,3);
s_int_variable(0x01,3);
s_int_variable(0x00,3);
s_int_variable(0x1b,3);

s_binary("00 01 02 00 1c 00 0c 03 00 28 00 04 ff 08 00 02");
s_int_variable(0x10,3);
s_int_variable(0x00,3);
s_int_variable(0x00,3);
s_int_variable(0x00,3);
s_int_variable(0x03);
s_string_variable("MSSQLServer");
s_binary("00 7c 09 00 00");
"mssql.spk" 25L, 547C written
```
How to run the SPIKE Script

```
[ Couldn't tcp connect to target
Fuzzing Variable 14:960
Couldn't tcp connect to target
Fuzzing Variable 14:961
Couldn't tcp connect to target
Fuzzing Variable 14:962
 Couldn't tcp connect to target
Fuzzing Variable 14:963
 Couldn't tcp connect to target
Fuzzing Variable 14:964
 Couldn't tcp connect to target
Fuzzing Variable 14:965
 Couldn't tcp connect to target
Fuzzing Variable 14:966
 Couldn't tcp connect to target
Fuzzing Variable 14:967
 Couldn't tcp connect to target
Fuzzing Variable 14:968
 Couldn't tcp connect to target
Fuzzing Variable 14:969
 Couldn't tcp connect to target
Done.
[dave@www3 src]$ ./generic_send_tcp 192.168.1.100 1433 audits/MSSQL/mssql.spk 0
```
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

  - 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

Inetinfo

Running as IUSR

RPC

Dllhost.exe

IWAM Token

This Admin token is sometimes sitting around

Admin Token

Overflow occurs in thread with IWAM token

Improper cloaking means dllhost can impersonate system!
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?
CANVAS

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 Professional 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
Other Immunity Products

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

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

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

**HashDB**  

**UNMASK**