VIRUS ANALYSIS 1

Code Red Buffer Overflow

Bruce McCorkendale and Péter Ször
Symantec Corporation

[Having encountered conflicting information from a variety of sources about the Code Red (aka W32/Bady.worm) buffer overflow technique, Bruce McCorkendale and Péter Ször decided to look into it themselves. Here, as a follow-up to Costin Raiu's analysis in last month's issue, they present their findings - Ed.]

Analyses of the Code Red worms to date have either skipped over the details of the buffer overflow, or they have given incorrect details. Noticing this, we were inspired to dig into the buffer overflow to uncover the details.

Setup

The IIS web server receives GET /default.ida? followed by 224 characters, URL encoding for 22 Unicode characters (44 bytes), an invalid Unicode encoding of %u00=a, HTTP 1.0, headers and a request body.

For the original Code Red worm, the 224 characters are N; for the most recent version of the worm, they are X. In all cases, the URL encoded characters are the same (they look like %uXXXX, where X is a hex digit – they have been published in previous analyses). The request body is different for each of the variants.

IIS keeps the body of the request in a heap buffer (probably the one it read into after processing the content-length header indicating the size to follow). Note that, despite the fact that a GET request is not allowed to have a request body, IIS dutifully reads it according to the header's instructions.

Buffer Overflow Details

While processing the 224 characters in the GET request, functions in IDQ.DLL overwrite the stack at least twice – once when expanding all characters to Unicode, then again when decoding the URL escaped characters. (The original *eEye* exploit demonstration probably uses one of these previous overwrites, but we have not seen this. *eEye* claims that they are overwriting the return address, which suggests that control is transferred to their shellcode when a RET instruction is executed.) However, the overwrite that results in the transfer of control to the worm body happens when IDQ.DLL calls DecodeURLEscapes() in QUERY.DLL.

The caller is supposed to specify a length in wide chars, but instead specifies a number of bytes. As a result, DecodeURLEscapes() thinks it has twice as much room as it actually has, so it ends up overwriting the stack. Some of

VIRUS BULLETIN ©2001 Virus Bulletin Ltd, The Pentagon, Abingdon, Oxfordshire, OX14 3YP, England. Tel +44 1235 555139. /2001/$0.00+2.50
No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the publishers.

the decoded Unicode characters specified in URL encoding end up overwriting a frame-based exception block. Even after the stack has been overwritten, processing continues until a routine is called in MSVCRT.DLL. This routine notices that something is wrong and throws an exception.

Exceptions are thrown by calling the KERNEL32.DLL routine RaiseException(). RaiseException() ends up transferring control to KiUserExceptionDispatcher() in NTDLL.DLL. When KiUserExceptionDispatcher() is invoked, EBX is pointing to the exception frame that was overwritten. (That EBX is pointing here is a side-effect of the immediately previous processing.) The exception frame is composed of four DWORDs, the second of which is the address of the exception handler for the represented frame.

The URL encoding whose expansion overwrote this frame starts with the third occurrence of %u9090 in the URL encoding, and is as follows:

%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3.

This decodes as the four DWORDs: 0x68589090, 0x7801CBD3, 0x90909090 and 0x00C38190. The address of the exception handler is set to 0x7801CBD3 (second DWORD), and KiUserExceptionDispatcher() calls there with EBX pointing at the first DWORD via CALL ECX.

The instruction CALL EBX is at address 0x7801CBD3 in MSVCRT.DLL. When KiUserExceptionDispatcher() invokes the exception handler, it calls to the CALL EBX, which, in turn, transfers control to the first byte of the overwritten exception block. When interpreted as code, these instructions find and then transfer control to the main worm code, which is in a request buffer in the heap.

The author of this exploit needed the decoded Unicode bytes to function both as the frame-based exception block containing a pointer to the 'exception handler' at 0x7801CBD3, and as runable code. The first DWORD of the exception block is filled with four bytes of instructions arranged so that they are harmless, but also place the 0x7801CBD3 at the second DWORD boundary of the exception block. The nop, nop, pop eax, push 7801CBD3h accomplish this task easily.

Having gained execution control on the stack (and avoiding a crash while running the 'exception block'), the code finds and executes the main worm code.

This code knows that there is a pointer (call it pHeapInfo) on the stack 0x300 bytes from EBX's current value. At pHeapInfo+0x78, there is a pointer (call it pRequestBuff) to a heap buffer containing the GET request's body, which contains the main worm code. With these two key pieces of information, the code transfers control to the worm body in the heap buffer. The worm code does its work, but never returns – the thread has been hijacked (along with the request buffer owned by the thread).

Conclusion

This technique of usurping exception handling is complicated (and crafting it must have been difficult). The brief period between the release of the *eEye* description of the original exploit and the appearance of the first Code Red worm leads us to believe that this technique is somewhat generic. Perhaps the exception handling technique has been known to a few buffer overflow enthusiasts for some time, and this particular overflow presented the perfect opportunity to use it.

Having exception frames on the stack makes them extremely vulnerable to overflows. This is an oversight in the current OS implementations, but *Windows XP* provides a new 'Vectored Exception Handling' feature that could allow exception frame data to be kept on the heap (however, current compilers only use stack-based exception frames).

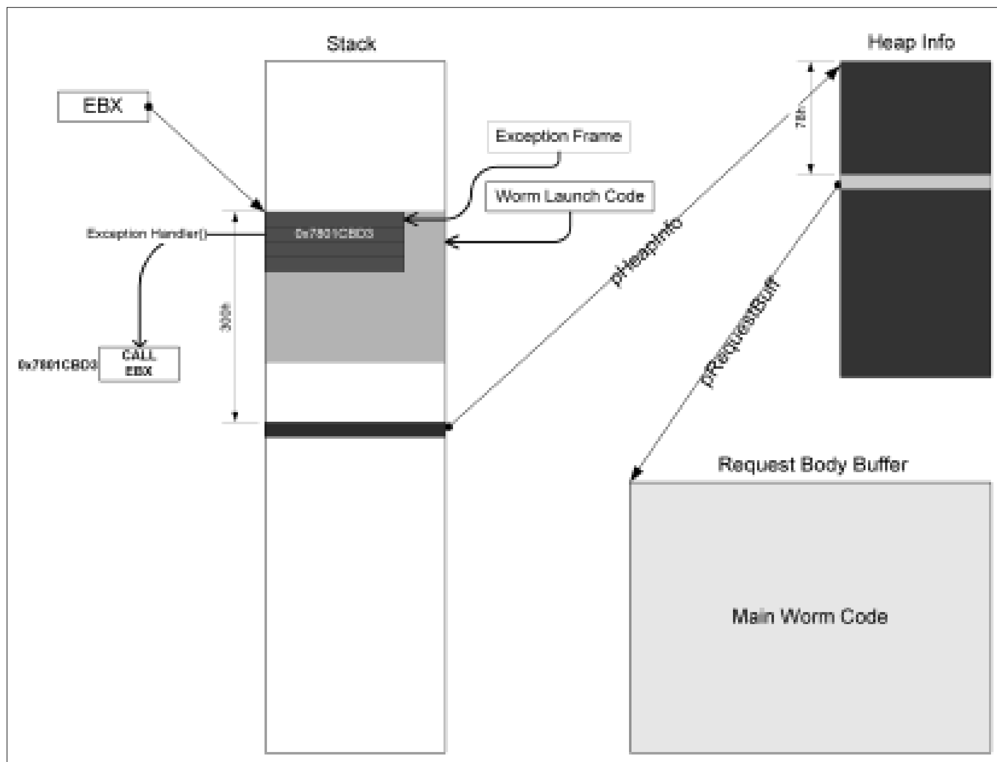

Figure 1: Stack, Heap, and Frame Layout.

VIRUS BULLETIN ©2001 Virus Bulletin Ltd, The Pentagon, Abingdon, Oxfordshire, OX14 3YP, England. Tel +44 1235 555139. /2001/$0.00+2.50
No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the publishers.