

[The MechMania 9 Cheating Experience]

--

By: Pavan Tumati (pavan@tumati.com) T.I.T. Department of Cheat Science,
Steve Hanna (shanna@uiuc.edu),
Contributions from: Mike Perry (mikeperry@fscked.org),
Chris Grier (grier@uiuc.edu)

(C) 2003, Pavan Tumati & Steve Hanna. All Rights Reserved. Document may be modified and reposted -if only if- the changes are clearly marked and a change list is inserted into a very visible location at the start of the document.

Message Table of Contents:

--

- 0) Planning
- 1) General Discussion / Server Structure
- 2) Packet Structures
 - a.) In process function interception
 - b.) LD_PRELOAD based interception
- 3) Crash/Exploit Techniques
 - a.) Generating Integer Overflow / Floating Bugs on the Server Side
 - b.) Encoding invalid indexes into packet structures that relied on Mech Unit IDs
 - c.) Causing the server to lose synchronization on the incoming data stream
- 4) Potential Attack Vector Analysis
 - a.) Crashing other clients via TrashTalk
 - b.) Warp/Fast Movement
 - c.) Money hacking
 - d.) Healing ourselves
- 5) Software Tools Developed and Utilized
- 6) End Summary
- 7) Credits

Planning

--

Exploitation and cheating has one facet that can cause the entire project to go awry or execute as planned; this stage is planning. One may think that this step is not vital but it should be clear that this stage can make or *break* any efforts that one may have. Quite a lot of time was spent planning for the MechMania Competition. From the original mention of the event, the planning started and the diabolical ideas began to form.

The initial collection of information began when I (Steve Hanna) was sent on reconnaissance missions into the ACM office. This stage allowed us to collect information and decide our plan of attack. From listening to network traffic and overhearing the conversations regarding MechMania we discovered that the server was written in Python. In addition to learning about the language of choice of the server we I

was able to overhear some problems that they were currently having with the server; these words of frustration inspired hope regarding our exploitation.

From this stage, various tools were collected (Decompyle, etc.) and past Python coding examples of the authors of MechMania were analyzed. Although, we were not able to gather extensive information this phase allowed us to feel out the undiscovered territory in the days to come.

Throughout the planning stage communication is vital. All members of the team were aware of the actions taking place and the data that was collected was shared among all members of the group. It was important to bounce ideas off of each other to determine the most effective tactics.

The bulk of the planning was done after the MechMania API was acquired. From the time that the team received the documentation we started picking apart possible code flaws that could be exploited. We reserved the back of the Green Street Coffee House and started dictating any free flowing thoughts into note form. Here the bulk of the ideas flowed forth. We examined each function declaration and questioned how this code could be used to our advantage. It should be noted that exploit coding is not a well defined science, it requires a little imagination to inspire the creative coding genius in us all (in this case the mad, coding zealot: Pavan). The important point that should be noted here is that every idea was written down and most possible exploiting techniques were taken into consideration. We kept in the back of our minds that it would merely take a small flaw to catapult our team into the winning ranks. In the pages to come you will learn how we dissected the MechMania protocol and attempted to take our team to victory.

General Discussion

--

SigMil is a sig about cheating; i.e., computer security is the art of cheating. People write code, but "cheating" and getting the code to do things outside of their intended use is what we study in SigMil. So, when the opportunity came up for us to test out our real-time cheating skills, we hopped at the chance. That opportunity was Mechmania. We had a great time, found a few server crashes, but had an otherwise great time studying and peering into the minds of the MechMania coders. It is evident to us that these guys do in fact use their minds, and we take our black hats off for them and thank them for the experience.

We spent a lot of time trying to cheat, but the guys who coded the server did an excellent job of guarding against most attacks that could be turned into an unfair advantage. They designed the game properly in that the clients only really -mirrored- copies of the data that were on the server. They did their job the right way, and did not code something flawed or broken. Under no circumstances (at least, in our analysis of their code) did they actually send any information that was not cross-checked against

server-side data. Overall, their design was simple, and it worked.

In this document, we'll discuss some of the techniques we used to study/analyze and work with the MechMania code. Steve Hanna and I (Pavan Tumati) were the guys who did the primary reverse engineering work; however, neither of us are Python programmers, so we taught ourselves pieces of the language as we went through. This slowed down our cheating efforts considerably, but we kept at it. We managed to decode the *entire* client/server protocol, as well as do a full exploit analysis of the code during the contest.

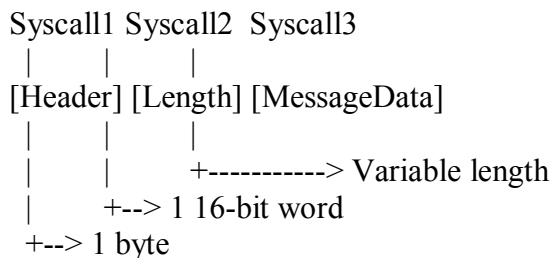
Most of the attacks against the server for cheating were guarded against; however, we did have the ability to cause it to crash through generation of exceptions, and by tweaking packet data before system calls were issued. We had the code analyzed and understood in approx 6-7 hours. (This is for you future cheaters with time management issues, if you plan on cheating like we did. Cheating during the contest is equally as difficult as coding a client, so it took a lot of fast coding and quick analysis. Incidentally, it's about as much fun as playing the game itself.)

2) Packet Structures

--

During the contest, we wrote a lot of tools to do packet analysis and cheat. The packet format for data transmission during the contest was:

(Figure 1)



Usually, though, the packet data from the client was not transmitted as 1 byte; instead, Dan Sledz' code made 3 individual system calls to transmit the individual pieces of data. During the contest, we observed this through strace; post contest, ethereal confirmed our results.

The header field was 1 byte, the length was 16-bits, and the message data was variable in length. Usually, the message data had some form of Whenever Dan's client-side code executed, he basically actually made 3 individual calls that dropped down into write system calls. We analyzed

this behavior through watching system call execution via strace. (strace and its usefulness are documented, I think, in the RevEng tutorial, found here: <http://www.acm.uiuc.edu/sigmil/RevEng/>)

The example packet transmission for team identification is shown here:

(Figure 2) - strace output

```
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(4000),
sin_addr=inet_addr("127.0.0.1")}, 16) = 0
write(3, "\200", 1)           = 1
write(3, "\0\5", 2)         = 2
write(3, "\1Don\0", 5)      = 5
```

2.a.) In process function interception

We figured out the various packet headers and formats by doing in-process function call interception, as well as through other various means. If you're curious how we were able to trace libc function, look at the attachment. That's a sample of the techniques we used -- the code we attached above doesn't actually work on Dan Sledz' code, because he, for some unknown reason, opens a socket at some point and then proceeds to use some form of unbuffered write operation on the socket descriptor.

We used a few general techniques to intercept the data:

- 1) Intercept socket() function call. I believe there was only 1 socket opened to connect the client to the server. We made a note of the file descriptor that was returned.
- 2) Intercept all read/write/send/recv operations. If they happened to be working on the socket descriptor that was returned to the process, we would then write the data to files, or modify the data in-stream to observe the effects on the server.

We created a singleton class to abstract away the details of stealing data:

(Figure 3)

```
-- BEGIN CODE SNIPPET --
class SocketDataInterceptor
{
public:
    static SocketDataInterceptor *Instance( void );
    static void Destroy( void );
```

```

~SocketDataInterceptor();
private:
struct PLTEEntry
{
    unsigned char jmpopcode __attribute__((packed));
    unsigned char modrm __attribute__((packed));
    unsigned int jumpaddress __attribute__((packed));
};

SocketDataInterceptor();
inline void ForcePLTEEntryCreation( void );

static int SDISend( int s, const void *msg, size_t len, int flags )
    __attribute__((cdecl));
static int SDIRecv( int s, void *buf, size_t len, int flags )
    __attribute__((cdecl));

PLTEEntry *m_SendPLTEEntry, *m_RecvPLTEEntry;
static SENDFUNCTION *m_SendFunction;
static RECVFUNCTION *m_RecvFunction;
static unsigned m_SDISendAddress, m_SDIRecvAddress;
static SocketDataInterceptor *m_Instance;
static ofstream m_BinarySendLog, m_BinaryRecvLog;
};
-- END CODE SNIPPET --

-- BEGIN CODE SNIPPET --
ForcePLTEEntryCreation();

m_SendPLTEEntry = (PLTEEntry *) dlsym( prochandle, "send" );
if( m_SendPLTEEntry == NULL )
    throw new DLError( "Unable to resolve send symbol" );
m_RecvPLTEEntry = (PLTEEntry *) dlsym( prochandle, "recv" );
if( m_RecvPLTEEntry == NULL )
    throw new DLError( "Unable to resolve recv symbol" );

// Unprotect and modify
ecode =
    mprotect( (void *) ((unsigned)( m_SendPLTEEntry ) & ~(pagesize-1) ),
        pagesize, PROT_READ | PROT_WRITE | PROT_EXEC );
if( ecode < 0 )
    throw new DLError( "Unable to unprotect region for send" );
m_SendFunction =
    (SENDERFUNCTION *) *((unsigned *) m_SendPLTEEntry->jumpaddress);
m_SendPLTEEntry->jumpaddress = (unsigned) &m_SDISendAddress;

```

-- END CODE SNIPPET --

Basically, we forced creation of PLT entries early on in execution, then we modified the PLT tables to serve as a trampoline to bounce functions to our static member methods that later recorded and analyzed data for us. In order to write our alternative jump locations, we unlocked page protections on the pages in which the PLT was stored, and then wrote in values that allowed us to redirect the function calls.

So what happened when we tried this initially? Dan Sledz threw a monkey wrench our way, and didn't actually call send/recv. This baffled us. What was he doing? We immediately peered into libmm9.so:

(Figure 4)

```
[gotcha]$ nm -C libmm9.so
<snip>
00007264 d p.0
    U pow@@GLIBC\_2.0
    U puts@@GLIBC\_2.0
    U read@@GLIBC\_2.0
    U realloc@@GLIBC\_2.0
    U select@@GLIBC\_2.0
    U signal@@GLIBC\_2.0
    U socket@@GLIBC\_2.0
    U sqrt@@GLIBC\_2.0
    U stderr@@GLIBC\_2.0
    U strcpy@@GLIBC\_2.0
    U strerror@@GLIBC\_2.0
    U strlen@@GLIBC\_2.0
00007280 d updsze
    U write@@GLIBC\_2.0
```

Glancing over dependencies, it appeared that he didn't use send/recv, but did in fact depend on socket/select and such. We needed to adapt the above code to tag write operations especially.

After much tracing in gdb, we determined that the actual writing of data was in a function called mm9writereal:

(Figure 5) Callstack Trace for mm9 Libraries before they hit the appropriate system calls

```
(gdb) bt
#0  0x420d1d10 in write () from /lib/tls/libc.so.6
#1  0x4010fccc in mm9writereal (sock=5, buf=0x83, size=25) at
```

```

clientlib.c:185
#2 0x40110023 in mm9sndmsg (sock=5, msg=131 '\203', size=25, buf=0x5
<Address 0x5 out of bounds>) at clientlib.c:223
#3 0x401100fb in mm9sendmsg (gam=0x5, msg=0x0) at clientlib.c:263
#4 0x401103c6 in mm9sendandresp (gam=0x19, msg=0x5) at clientlib.c:358
#5 0x40112f06 in mm9build (gam=0x804a208, id=0, prod=0xbffeda0) at
clientlib.c:1580
#6 0x4001b3fa in mm9::Base::build(mm9::MechStats) (this=0x804c040, stats=
{armor = 5, batt = 50, turnsToMove = 4, range = 5, damage = 3,
canHarvest = true}) at unit.cc:157
#7 0x080490be in DonTeam::userTurn() (this=0x804a6d8) at don.cxx:51
#8 0x4001cdcb in mm9::Game::gameLoop() (this=0x804a1f0) at game.cc:61
#9 0x4001cf98 in mm9::Game::startGame(mm9::Team*, std::string const&,
mm9::ClientType, unsigned short) (this=0x804a1f0,
team=0x0, host=@0xbffeff0, ctype=ClientClient, port=0) at game.cc:127
#10 0x0804954c in main (argc=1, argv=0xbffff064) at don.cxx:102
#11 0x420156a4 in __libc_start_main () from /lib/tls/libc.so.6

```

At this point, we just decided to switch over and write a quick LD_PRELOAD based function call interception system to trap his code. And that takes us to the next section: LD_PRELOAD based interception.

2.b) LD_PRELOAD based interception

A combination of the above techniques (in the attached code), and through use of the LD_PRELOAD functionality, we were able to hack/modify the client to transmit the data. Alternatively, we probably could've dlsym'd the __libc_write() function, but in the heat of competition, we just rapidly coded another tool.

Essentially, we had to build our own shared library. So, we created a shared object that implemented wrappers for the send() and write() functions. This library was used in conjunction with the LD_PRELOAD environment variable so that we could insert wrappers as needed. We added two rules to the makefile:

(Figure 6) Make file modifications

```

libintercept.so.1.0.1: Interceptor.o DLError.o
$(CC) -shared -Wl,-soname,libintercept.so.1 \
    -olibintercept.so.1.0.1 Interceptor.o DLError.o -lc -ldl

Interceptor.o: Interceptor.cpp DLError.cpp
$(CC) -fPIC -rdynamic -g -c -Wall Interceptor.cpp DLError.cpp

```

In the .cpp file, encapsulated in extern "C", we added two functions.

Their general structure was as follows:

(Figure 7) Socket Wrapper Function

```
int socket( int domain, int type, int protocol )
{
    void *handle;
    SOCKETFUNCTION *oldsocket;
    int ecode;

    printf( "Interception." );
    handle = dlopen( "/lib/libc.so.6", RTLD_LAZY );
    if( !handle )
        return -1;
    oldsocket = (SOCKETFUNCTION *) dlsym( handle, "socket" );
    if( oldsocket == NULL )
        return -1;
    ecode = (*oldsocket)( domain, type, protocol );
    if( !( ecode < 0 ) )
    {
        g_MonitorSocket = ecode;
    }

    return ecode;
}
```

Notice in Figure 7, that we saved the return value to a global variable. We did this because we determined that the program only opened one socket. We saved this value to detect when the write() function was being called in order to operate on this socket.

The write function was implemented as follows:

(Figure 8)

```
int write( int fd, const void *buf, size_t count )
{
    void *handle;
    WRITEFUNCTION *oldwrite;
    int ecode;

    handle = dlopen( "/lib/libc.so.6", RTLD_LAZY );
    if( !handle )
        return -1;
    oldwrite = (WRITEFUNCTION *) dlsym( handle, "write" );
    if( oldwrite == NULL )
```

```

return -1;
if( fd == g_MonitorSocket )
{
switch( g_MonitorState )
{
case HIJACK_IDINTERCEPT:
printf( "ID Intercepted: %02X\n", ((unsigned char *) buf)[0] );
g_MonitorState++;
break;
case HIJACK_LENGTHINTERCEPT:
printf( "Length intercepted: %04X\n",
((unsigned short *)buf)[0] );
g_MonitorState++;
break;
case HIJACK_DATAINTERCEPT:
{
unsigned *datapointer = (unsigned *) buf;
printf( "Attacker: %08X\nAttackee:%08X\nDamage: %08X\n",
datapointer[0], datapointer[1], datapointer[2] );
// datapointer[0] = -1;
// datapointer[1] = -datapointer[1];
g_MonitorState++;
}
break;
case HIJACK_IDLE:
break;
}
}
ecode = (*oldwrite)( fd, buf, count );
return ecode;
}

```

Notice that we have a little switch. In our code, we maintained a state machine to determine what was going on. This particular hackish write function was designed to analyze what happened on attack packets. It was possible, for example, to modify the data flowing through on attacks and actually -crash- the server by changing the Attacker/Attackee/Damage fields of the transmitted packet.

3) Crash/Exploit Techniques

There were a few ways we were able to crash the server in our code. In the previous section, we demonstrated one.

Here is a general summary of the techniques we used:

- 1) Generating integer overflow bugs and floating point errors on the server side
- 2) Encoding invalid indexes into packet structures that relied on unit IDs
- 3) Mishandling data streams

3.a) CRASH TECHNIQUE: Generating Integer Overflow / Floating Bugs on the Server Side

When creating mechs, it was possible to place negative values. When these values were then pulled off the wire, they were converted into unsigned integers with extremely large values. These values were then pumped into calculations that overflowed on the server side and generated exceptions.

The code in question was here (below, Figure 10). The values pulled off the wire are then reutilized in a `get_cost` function. This function can be toyed with, by tweaking parameters, to cause the server to crash.

(Figure 10)

```
def handle_build(stuff, sock, connections, obj_list, create_list):
    baseid = None
    try:
        (baseid, armor, battery, speed, rng, damage, can_harvest,) =
            struct.unpack('!IIIIIB', stuff['value'])
    except:
        if debug:
            print( 'sending badmesg to %s because invalid build packet' %
                util.getName(sock, connections))
            util.send_badmesg(sock)
        if (baseid != None):
            dicti = { 'armor': armor,
                    'battery': battery,
                    'speed': speed,
                    'range': rng,
                    'damage': damage,
                    'can_harvest': can_harvest }
            cost = util.get_cost(dicti)
            conn_index = map(lambda x:x.sock, connections).index(sock)
            team = connections[conn_index].team
            if (not util.checkperm(obj_list[baseid], connections, sock)):
                if debug:
                    print 'sending badperm because building using wrong base'
                    util.send_badperm(sock)
                elif (cost > team.money):
                    if debug:
                        print( 'Sending badmesg to %s because cost is > money' %
```

```

        util.getName(sock, connections))
    util.send_badmesg(sock)
elif (cost == -1):
    if debug:
        print ('Sending badmesg to %s because of attribute values' %
            util.getName(sock, connections))
        util.send_badmesg(sock)
elif (cost > mesg.rules['max_point_cost']):
    if debug:
        print( 'Sending badmesg to %s because mech is too good' %
            util.getName(sock, connections))
        util.send_badmesg(sock)
elif (obj_list[baseid].eta == -1):

```

(Figure 11) Get Cost Function

```

def get_cost(attrs):
    if ((attrs['armor'] < 0) or ((attrs['battery'] < 0) or
((attrs['range'] < 0) or ((attrs['damage'] < 0) or ((attrs['speed'\
] <= 0) or (not (attrs['can_harvest'] in [0,
1]))))))):
        return -1
    else:
        return int(((((((attrs['can_harvest'] *
mesg.rules['cost_to_harvest']) + (attrs['armor'] /
mesg.rules['armor_per_poin\
t'])) + (mesg.rules['speed_cost_for_one_turn_move'] / attrs['speed'])) +
(attrs['battery'] / mesg.rules['batt_per_point'])) +\
(((attrs['range'] + mesg.rules['min_range_cost']) * (attrs['damage'] +
mesg.rules['min_damage_cost'])) * mesg.rules['rangeda\
m_mult'])) + 0.5))

```

3.b) CRASH TECHNIQUE: Encoding invalid indexes into packet structures that relied on Mech Unit IDs

In the previous section (that is, section 2) we demonstrated in source code what would happen if we encoded negative indexes into the packet. The real reason this worked is because on the server side, this is the code (below) that was executed. Notice that the attacker, attack, and damage values are pulled straight off the wire and used as integer indexes. The 'I' character pulls a 32-bit unsigned value; a negative value causes an invalid index to be utilized. When this happens, the server crashes with an exception and dies.

(Figure 12)


```
util.send_badmesg(sock)
```

3.d) CRASH TECHNIQUE: Causing the server to lose synchronization on the incoming data stream

It was also possible to make the server mishandle its data stream by using undefined byte-codes in the header of the packet (I think -- we didn't have time to test this or look into it for causing the server to malfunction), as seen here:

(Figure 13)

```
-- Main Handling Block in Server Code
Note: Python is whitespace sensitive, I've realined things
--
if (not (stuff['type'] in range(mesg.MSG_MOVE, (mesg.MSG_ENDTURN + 1)))):
    if debug:
        print ('Sending badmesg to %s because invalid message' %
              util.getName(sock, connections))
            util.send_badmesg(sock)
    elif (stuff['type'] == mesg.MSG_MOVE):
        servfunc.handle_move(stuff, sock, connections, obj_list, themap, move_list)
    elif (stuff['type'] == mesg.MSG_ATTACK):
        servfunc.handle_attack(stuff, sock, connections, obj_list, attack_list)
    elif (stuff['type'] == mesg.MSG_BUILD):
        servfunc.handle_build(stuff, sock, connections, obj_list, create_list)
    elif (stuff['type'] == mesg.MSG_TRASH):
        servfunc.handle_trash(stuff, connections, sock, canwin )
    elif (stuff['type'] == mesg.MSG_ENDTURN):
        conn_index = map(lambda x:x.sock, connections).index(sock)
            connections[conn_index].turn_ended = 1
            num_clients_remaining -= 1
            util.send_ack(sock)
```

Since there are no other clauses here, it is possibl

... No else block for other message types? It's possible to cause the loop to break here.

```
--
```

So, during interception/modification of the write function calls, we had to write a state-machine to keep track of what phase of interaction the client was going through. On the server side, the server also pulled data from the socket in pieces.


```

util.send_msg( connections[conn_index].sock, mesg.MSG_SERVETRASH,
              (struct.pack('!I', teamnum) + msg))
except:
connections[conn_index].closed = 1
if (conn_index in canwin):
    if debug:
        print ('Removing %s from canwin because io error on trash send' %
              connections[conn_index].team.name)
    canwin.remove(conn_index)
    if (connections[conn_index].team.type == mesg.CLIENT_CLIENT):
        num_clients_remaining -= 1
    continue

```

(Figure 15)

```

def send_msg(sock, typ, msg):
    length = len(msg)
    if (length > 65535):
        raise 'TooLongAMessage'
    sendstr = (pack('!BH', typ, length) + msg)
    if (not (type(sock) == type([]))):
        sock.send(sendstr)
    else:
        for sck in sock:
            sck.send(sendstr)

```

Idea - Warping/Fast Movement

--

We looked into this from the server side. The server was intelligent about this, and computed distance values based on mech-object data stored in previous data structures.

Here's the code that needed to be analyzed:

(Figure 14)

```

-- BEGIN CODE SNIPPET --
def handle_move(stuff, sock, connections, obj_list, themap, move_list):
    id = None
    try:
        (id, movex, movey,) = struct.unpack('!III', stuff['value'])
    except:
        if debug:

```

```

print ( 'Sending badmesg to %s because invalid move packet' %
        util.getName(sock, connections) )
util.send_badmesg(sock)
if (id != None):
if (not util.checkperm(obj_list[id], connections, sock)):
    if debug:
        print ( "Permission denied: %s tried to move other team's mech" %
                util.getName(sock, connections))
        util.send_badperm(sock)
elif (obj_list[id].hp <= 0):
    if debug:
        print ( '%s attempting to move dead mech' %
                util.getName(sock, connections))
        util.send_badmesg(sock)
elif( util.canmove(obj_list[id].position, (movey,movex), themap) and
        (obj_list[id].type == mesg.MECH_TYPE)):
    move_list.append((id, (movey, movex) ) )
    util.send_ack(sock)
else:
    if debug:
        print ( "Sending badmesg to %s because can't move, or not a mech" %
                util.getName(sock, connections))
        util.send_badmesg(sock)
-- END CODE SNIPPET --

```

Notice that Andrew Lusk checks object permissions as the first thing. This was smart; we glanced over the code and determined that ownership of an ID was cross-checked against the actual socket descriptor number. That is, if the request for changing a unit came in over the socket, that unit had to have been owned by the team that was connected over over -that specific socket-. Secondly, the server checked to see if the unit was alive. The real function that protected warping was the `util.can_move` function; that function checked the distances and made sure that the movement was happening within a valid range:

(Figure 15)

```

-- BEGIN CODE SNIPPET--
def canmove(pos1, pos2, themap):
if ( (pos2[0] < 0) or ((pos2[1] < 0) or ((pos2[0] >= len(themap)) or
    (pos2[1] >= len(themap[0])))):
    if debug:
        print 'outside of map dimensions'
        return 0
dist = ((pos1[1] - pos2[1]),
        (pos1[0] - pos2[0]))

```

```

    if ((abs(dist[0]) > 1) or ((abs(dist[1]) > 1) or ((dist[0] *
dist[1]) == -1))):
        if debug:
            print 'too far away'
        return 0
    if (themap[pos2[0]][pos2[1]].type <= 0):
        if debug:
            print ('blocked terrain at %s' % str(pos2))
        return 0
    if (len(themap[pos2[0]][pos2[1]].objs) > 3):
        if debug:
            print 'too crowded!'
    return (len(themap[pos2[0]][pos2[1]].objs) < 4)
-- END CODE SNIPPET--

```

So, at a cursory glance, it didn't seem as if we could hack this function.

Idea - Money hacking

--

The server maintained most of the information about money. After processing all the events that it learned about, it simply send packets updating the money fields. There was no real room to hack money making from the client side, ... at least, that we know of. The server periodically updated money values and sent the values back. The code is not particularly interesting, so it has been omitted from this document.

Idea - Healing Ourselves

--

The server doesn't actually check to see if the damage value sent with the creation of mechs is negative. Since we were unfamiliar with python, we looked into this possible bug into great detail. It never panned out, because when the `unpack()` function was executed, Andrew explicitly specified that he wanted an unsigned integer, so the data values sent across the wire ended up being promoted into long integers, and the attack calculations failed because we'd always get insufficient battery.

Amusingly, though, we were misled for quite some time because we didn't realize 'I' in `unpack` was for unsigned integers. I suppose it would be interesting if we could see what would happen if the 'I' command was not used instead.

(Figure 16)

```

def handle_attack(stuff, sock, connections, obj_list, attack_list):
    attacker = None
    try:
        (attacker, attackee, damage,) = struct.unpack('!III', stuff['value'])
    except:
<HUGE SNIP>
        distance = util.getdist( obj_list[attacker].position,
                                obj_list[attackee].position )
        battery = util.get_battery( obj_list[attacker], distance, damage )
<HUGE SNIP>

```

(Figure 17)

```

def get_battery(obj, rng, dam):
    return int((((piecewise(rng, obj.range) * piecewise(dam, obj.damage))
* msg.rules['normal_attack_battery']) + 0.5))

```

(Figure 18)

```

def piecewise(val, rated):
    if (val <= 0):
        val = 0
    if (rated <= 0):
        if (val == 0):
            return 1.0
        else:
            return 100.0
    if (val <= rated):
        return ((1.0 / 3.0) + ((2.0 * val) / (3.0 * rated)))
    else:
        if ((val / rated) > 3.5):
            return 100.0
        ret = (math_e ** (((val - rated) * 2.0) / rated))
        if (ret > 100.0):
            return 100.0
        return ret

```

5) Software Tools Developed and Utilized

We used these tools, and a few others in the process. If you're interested, you should investigate the usage of the tools.

- SocketDataInterceptor, Inline process function interceptor, Source Code Attached [in the future]
- libintercept, LD_PRELOAD library function interceptor Source Code Attached [in the future]

- decompyle, Python Bytecode Disassembler
- Ethereal, Network packet analyzer
- strace, System Call Tracer
- ltrace, Library Call Tracer
- objdump, Section Dumper/Disassembly Tool
- nm, List Symbols from Object Files
- gdb, --set-disassembly-flavor intel, disassemble

6) End Summary

Overall, we think our efforts were more entertaining than anything else. The server structure prevented us from turning any flaws into advantages. Still, we believe the general use of these techniques can assist future cheaters and programmers in developing tools quickly for analysis of data streams during contests such as MechMania. We hope that by reading this document and reading about our thought process will assist readers in finding new and interesting tricks to "enhance game play."

7) Credits

- Steve Hanna, for letting me use his box as root. I didn't do anything evil, I swear.
- Chris Grier & Mike Perry, for being on the team, and showing me some new evil tricks
- The MechMania team, for creating the event and giving us a legit reason to lose sleep
- All rodents in the world, since clearly rodentia is far superior to any other class of animals.

This document:

(C) 2003, Pavan Tumati & Steve Hanna. All Rights Reserved. Document may not be duplicated, modified, or posted without the express written consent of Pavan Tumati.