



Cisco IOS Shellcodes

Gyan Chawdhary, Senior Consultant
Varun Uppal, Senior Consultant



Agenda

- Background and research aims
- Worked example
 - IOS Debugging
 - IOS Shellcode Development Tools
 - Building IOS Shellcodes
 - Bypassing Checkheaps()
 - Potential Impact and Threat Scenarios
- Mitigation and conclusions



Why Investigate IOS

- Very little is known about the tools/techniques used by Lynn to create IOS based shell codes
- IOS security is often overlooked in favor of OS/Host based security
- To demonstrate and reiterate that Cisco shell codes are possible and not difficult to write
- To identify mitigating factors for any issues or loop holes found in IOS



Introduction to IOS

- Monolithic Architecture – one big ELF file
- Everything is tightly integrated, and non modular
- Virtual memory scheme not fully implemented, has a flat memory model
- Uses stack and heap data, however everything including stack is stored in heap 😊



IOS Debugging



Decompressing the IOS Firmware image

- IOS uses a modified pkzip format for image compression
- The IOS boot loader unzips the image at runtime
- Tools – Standard Unix “unzip”
 - Stffit Expander
 - WinRar



Fixing the ELF Header (1)

- The uncompressed IOS firmware is a standard ELF image
- The ELF header is slightly modified to prevent attackers from reverse engineering the image offline

```
typedef struct
{
    unsigned char    e_ident[EI_NIDENT];    /* Magic number and other info */
    Elf32_Half       e_type;                /* Object file type */
    Elf32_Half       e_machine;             /* Architecture */
```



Fixing the ELF Header (2)

- As we are working with a PowerPC based Cisco router (2600) we use this as the e_machine id for the target Architecture
- Using a hex editor, change the “e_machine” bit to 0x14
- The image will now load in IDA using PPC instruction set



Setting up GDB (1)

- GDB – The GNU Debugger
 - IOS contains a GDB stub with limited functionality
 - Must be connected via a serial port
 - Version 6.0 was the last version to support IOS
 - Several tweaks to the GDB source required in order to correctly print addresses which would otherwise cause problems, especially while writing shellcode to target memory addresses using the gdb “set” command



Setting up GDB (2) – Editing config.bfd

- `powerpcle-*-solaris2* | powerpcle-*-linux-* | powerpcle-*-vxworks*) targ_defvec=bfd_elf32_powerpcle_vec
targ_selvecs="rs6000coff_vec bfd_elf32_powerpc_vec
ppcboot_vec" targ64_selvecs="bfd_elf64_powerpc_vec
bfd_elf64_powerpcle_vec"`
- `powerpcle-*-solaris2* | powerpcle-*-linux-* | powerpcle-*-vxworks*) targ_defvec=bfd_elf32_powerpcle_vec #
targ_selvecs="rs6000coff_vec bfd_elf32_powerpc_vec
ppcboot_vec" targ64_selvecs="bfd_elf64_powerpc_vec
bfd_elf64_powerpcle_vec"
targ_selvecs="cisco_core_big_vec ieee_vec"`



Setting up GDB (3) – Editing remote.c

Comment out the following lines:-

```
if (remote_cisco_mode == 0)          /* or declare global as 1 */
{
    c = readchar (remote_timeout);
    csum += c;
    repeat = c - ' ' + 3;    /* Compute repeat count */
}
else                                Cont ..
```



Setting up GDB (3)

- `/configure --target powerpcle-elf`
- DONE - we have a fully working command line IOS serial debugger with read, write and continue functionality.



IOS software development environment

- GDB – The GNU Debugger

```
(gdb) set processor powerpc-MPC8XX (using 2621XM)
```

```
(gdb) target remote /dev/ttyS0
```

```
(gdb) disass 0x83000000 0x8300000c
```

Dump of assembler code from 0x83000000 to 0x8300000c:

```
0x83000000:      lis      r3,0
0x83000004:      addi     r3,r3,56
0x83000008:      lis      r4,0
0x8300000c:      addi     r4,r4,60
```



IOS Shell code Development Tools



Tools

- We write the shell code in pure PPC assembler
- The GAS (GNU Assembler) is used to assemble the asm code
- Opcodes are then extracted using “objdump -d”
- We use a shell script to translate the objdump output to gdb “set” commands which can then be directly processed by GDB



Jump Vector Patch (1)

- In order to test our shell code we patch an existing IOS function to execute the shell code in memory
- This can be achieved by constructing a .gdbinit file which automatically patches the router with our jump vector.



Jump Vector Patch (2)

target remote /dev/ttyS0	—————>	Remote Serial Line
set *0x804A42D8=0x3d208312		
set *0x804A42DC=0x38093a20		
set *0x804A42E0=0x7c0903a6		
set *0x804A42E4=0x4e800421	—————>	jump to *shellcode
set *0x804A42E8=0x3d20804a		
set *0x804A42EC=0x380943f8		
set *0x804A42F0=0x7c0903a6	—————>	return(0)
set *0x804A42F4=0x4e800421		
source shellcode.txt	—————>	Shellcode Patch



IOS GDB script (1)

```
#!/bin/bash
file=${1}
base=${2}
#if [[ $# -ne 2 ]]; then
#    echo "usage ./go.sh <a.s> <base_add>"
#    exit
#fi
for i in `grep -A200 '0:' ${file} | sed 's/^\.*:\/0x\1/' | cut -f 1-4 -d ' ' |
sed 's/[[:space:]]//g`; do

    printf "set *0x%X=${i}\n" "${base}"
    base=$((base + 4))

done
```



IOS GDB script (2)

- The script takes a single base address as its argument, which will then be used to store the shell code in memory
- The script generates a set command list in the following format: `set * <where> * <what>` which will write the shell code in runtime router memory



Programming the IOS



IOS software development environment

- Hello World for IOS

```
.text
```

```
.equ printf,0x803C4800
```

```
.global start
```

```
start: bl      start2
```

```
.string "Hello world!!!\n"
```

```
start2: mflr    3          #address of string name is in LR
```

```
lis        7, printf@ha
```

```
addi       7,7,printf@l    #address of printf into CTR
```

```
mtctr      7
```

```
bctrl                      #call printf()
```



IOS software development environment

- Invoking IOS functions
- We will use *mtctr* and *bctrl* powerpc instructions to invoke functions under IOS
- This is similar to `__attribute__((longcall))` under C
- To transfer control to an IOS function, the Count Register is first loaded with the target address from a General-Purpose Register using the *mtctr* instruction
- The *bctrl* instruction is then called to branch to the Count Register, which has the address of our loaded API



IOS software development problems

Sample code:

lis	7, API@ha	#Load API address in r7
addi	7,7,APIf@1	#Load API address in r7
mtctr	7	#move r7 to count register
bctrl		#branch to count

- Using this method, we can invoke API's under IOS
- Example: listen(), connect(), bind()



Building IOS Shell Codes



Reversing the IOS

Step 1 – Uncompress the image, fix up the ELF header

Step 2 – Load the image in IDA

Step 3 – Wait forever (Use older IOS image (11.0) which loads faster)

Step 4 – Analysis: We use both static and runtime analysis tools to discover interesting functions in IOS for shell code development.



Reversing the IOS

Example:

- IOS Finger Command - Cisco supports a *finger* daemon to give information about who is connected to a router
- The output is similar to *show users* command when run locally on the system
- We log the output and associated strings, which are then searched in the main image using IDA.
- Functions are further mapped using break points and creating call graphs.



Other useful commands ..

- show memory
- show context
- reload (useful for mapping checkheaps() function)



The bind shell

- Four hard-coded addresses required
- Creates a new VTY
- Allocate memory for a command information structure
- Set a password on the VTY line
- Privilege escalate to “Level 15”



The bind shell

Setting the password:

- Command information structure + **0x0a68** = start VTY line
- Command information structure + **0x0a6c** = end VTY line
- Call `change_pass()` function

Escalating privileges:

- At a fixed address - array of pointers to VTY line structures
- We need **&Array[66]**
- Within this structure, at **0x0de4** is a password structure
- Set value to **0xff800000** – Level 15 😊



The bind shell

[illegible]

Click image to start

Please be patient – it takes a little while for the shellcode to start running

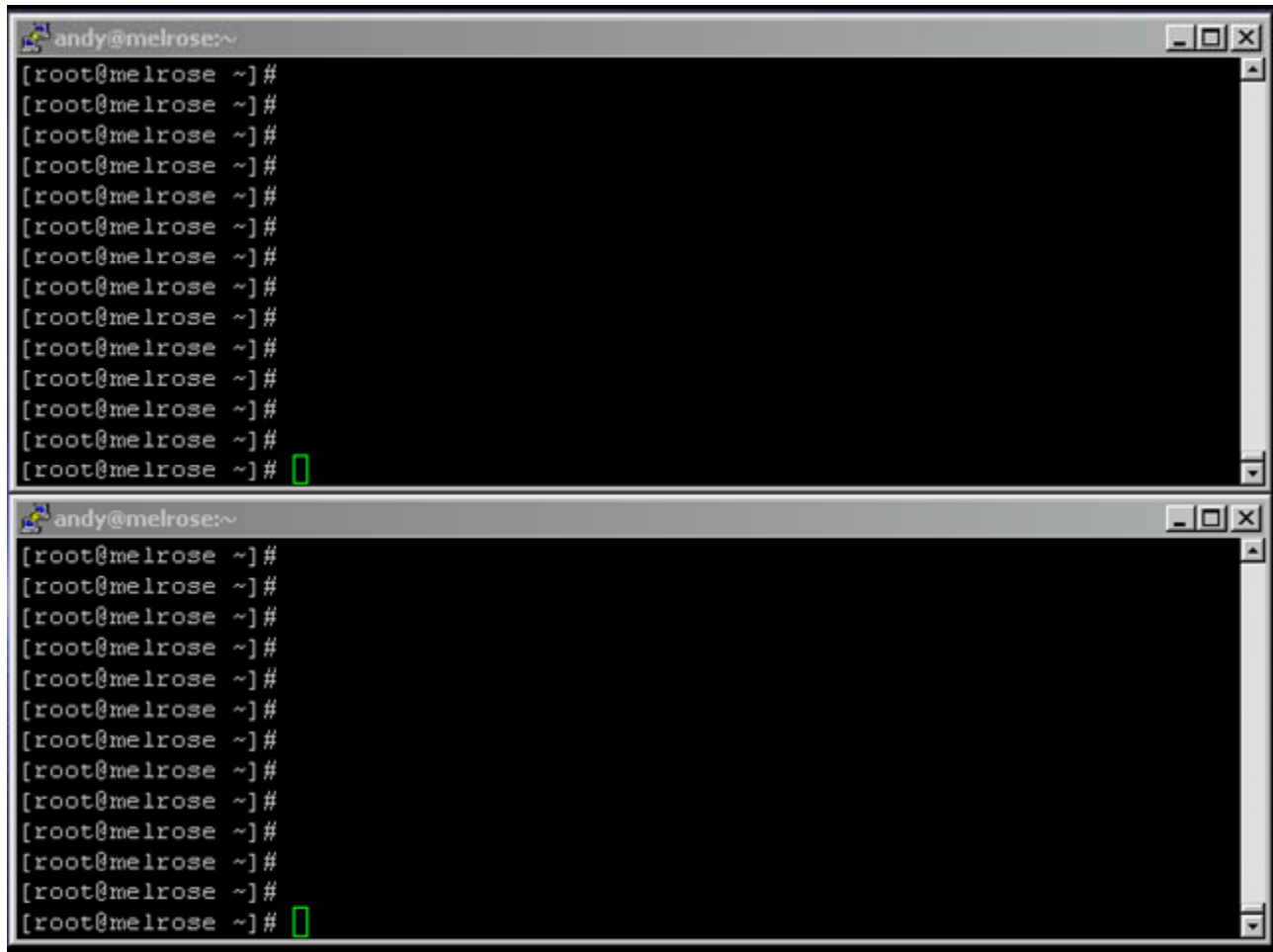


The reverse shell

- Five hard-coded addresses required
- Creates a new VTY
- Privilege escalate to level 15
- Opens a TCP connection
- Connects the VTY to the TCP connection



The reverse shell - Demo



Click image to start

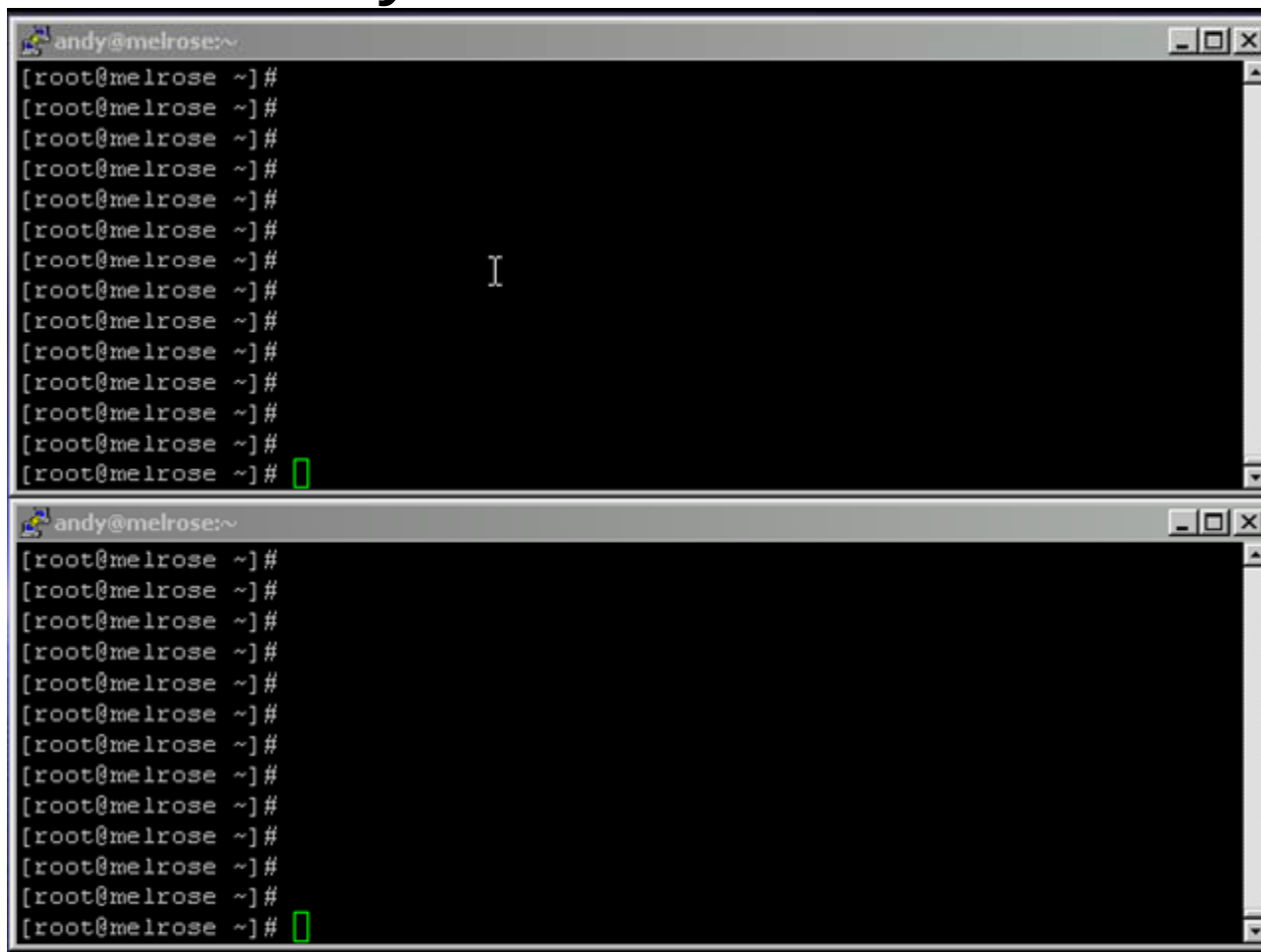
**Please be patient –
it takes a little
while for the
shellcode to start
running**



The “2 byte rootshell” – bindshell (v2)

- TTY Line structure – the structure that holds privilege level
- TTY Line structure + **0x0174** = **0x00000001**
- Set the LSB to zero and the router no longer prompts you for any authentication credentials 😊
- TTY Line structure + **0xde4** = **0x11800000**
- Set the MSB to **0xff** to escalate to level 15

The “2 byte rootshell” – Demo



```
andy@melrose:~  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#  
[root@melrose ~]#
```

Click image to start

Please be patient –
its takes a little
while for the
shellcode to start
running



The CheckHeaps() Issue



The checkheaps() issue

- *Checkheaps is a periodic process that verifies the sanity of the heap memory buffers (dynamic memory is allocated from the system heap memory region) and the integrity of the code region.*



Process Watch Dog

- Scheduler allocates a watch dog timer for each process
- Polls a process, if process runs > than preset period of 2 seconds the scheduler regains control and generates a warning
- If the preset expires a 2nd time, the watch dog fires a termination request against the rouge process



Process Watch Dog

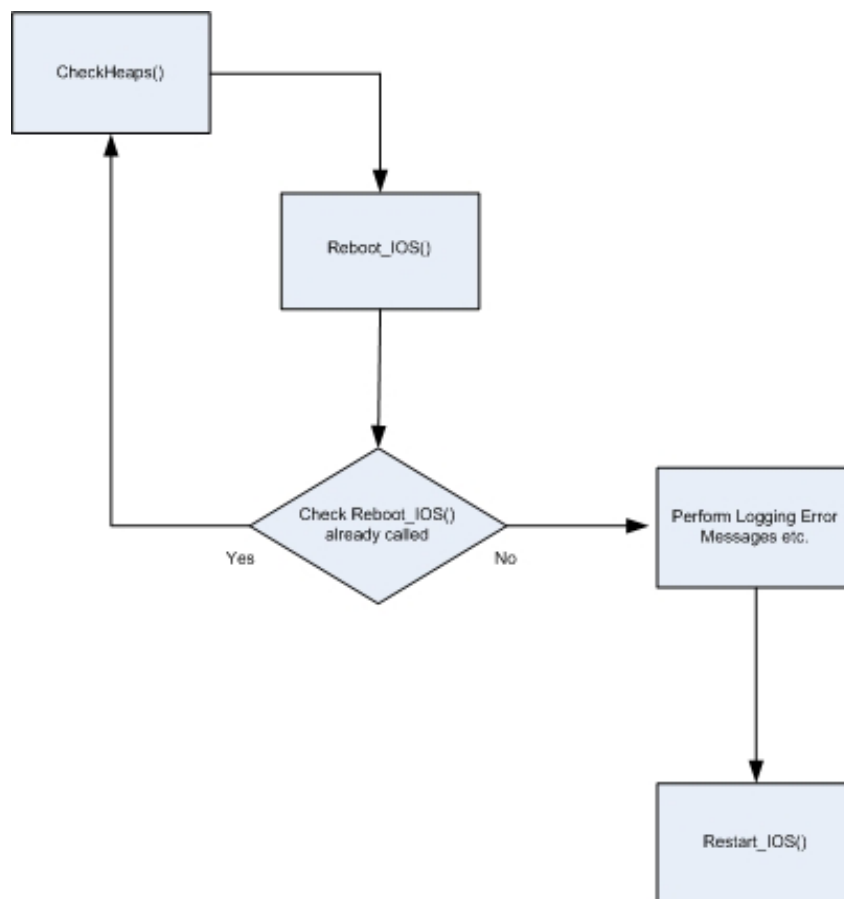
- IOS allocates process priorities to each process, Critical, High, Medium and Low
- Critical – Resource allocation processes
- High – Fast Packet switching processes
- Medium – Default
- Low – Check heaps, system management processes.
- Being a low priority process, check heaps is killed



A word on Timers

- Used by IOS for event scheduling, context switching etc
- Runs a Master timer, and n number of slaves based on the process
- All this information is managed using a timer Linked List
- This linked list can be abused to overwrite arbitrary memory locations

Bypassing checkheaps()





Bypassing checkheaps()

- First demonstrated by Michael Lynn at Black hat in 2005
- Might have taken advantage of the timer linked lists to overwrite the “crashing_already” flag
- Cisco simply fixed the timers issue vector, NOT the check heaps crashing_already bug



Potential Impact and Countermeasures



Potential Impact

- The process of building an IOS shell code can be automated
- IOS exploitation can be made 100% reliable when attacking internally.
- Stable memory resident backdoors can be created using the outlined techniques in this presentation



Countermeasures

- Keep the IOS firmware upto date – Not always feasible
- Close all unwanted services
- Apply ACL's and strong access control policy



Questions?