

The Holy Grail:

***Cisco IOS Shellcode And
Exploitation Techniques***

Michael Lynn

Internet Security Systems



 **INTERNET | SECURITY | SYSTEMS®**

Another Unbreakable System



Why You Should Care

■ **Wide Deployment**

- Switches
- Routers
- Access Points

■ **Keys To The Kingdom (MITM)**

- Control the network traffic
- Packet sniff in far off lands
- Modify traffic
- Break weakly authenticated encryption (passwords, etc)

Some Review: Basic Techniques

■ **Stack Overflows**

- Overwrite return address on the stack

■ **Heap Overflows (Pointer Exchange)**

- Traditionally we use heap chunk linkage
- Any linked list will do

Typical linked list delink looks like:

```
foo->prev->next = foo->next;  
foo->next->prev = foo->prev;
```

Misconceptions

- **Routers And Switches Are Just Hardware**
- **It Is Not Possible To Overflow Buffers On IOS**
- **There Is No Way To Exploit Buffer Overflows On IOS**
- **Every Router Is So Different That An Exploit Might Work On One Router But Never Another**

Wrong!

- **Routers And Switches Run Software On General Purpose CPUs**
- **Buffers Do Exist And It Is Not So Rare That They Overrun**
- **Exploitation Is Possible**
- **Exploitation Can Be Made Reliable And Cross Platform (more on this later)**

■ **Monolithic**

- No loadable modules (yet)
- All addresses are static
- All addresses are different per build

■ **Real Time OS**

- If you are running you own the CPU (mostly)
- We have to exit or yield properly or we will crash
- Once our code is running we have won any race

■ **Stability**

- IOS tends to favor rebooting over correcting errors

A Word On Code Quality

■ **Much Better Than Most Platforms**

- They check heap linkage
- They are very aware of integer issues
- They almost never use the stack
- They have a process to check all heaps
- Very old, very well tested code

■ **Bugs Exist Anyways**

- Green pastures
- We can get around some checks
- We will use some of these checks against them

The Dreaded Check Heaps Process

- **Walks All Heaps Looking For Bad Linkage**
 - Even if our chunk is not freed check heaps will detect bad linkage
 - Is run every 30 to 60 seconds depending on load
- **This Is The Main Reason Heap Overflows Can Be Hard**

Rules of Engagement

■ **Stack Overflows**

- Rare, but if we find one, its fair game

■ **Heap Overflows**

- They check next and previous pointers
- We either have to beat check heaps or not offend it
- We must either know the values for the previous pointer or we must get around this somehow

■ **Monolithic Architecture**

- For heap overflows we must have exact offsets per version (more on this later)

A Look At IOS Heap Structures

- We Can't Overflow Past Next Pointer
- We Can't Overwrite Magic Number
 - Magic Number is 0xAB1234BC
- We Can't Overwrite Red Zone
 - Red Zone value is 0xFD1001DF



■ His Previous Presentations

- Blackhat 2002
- Defcon X

■ His Technique

- Uncontrolled pointer exchange (more on this later)
- Flash invalidating
- Guessing previous pointer

■ His Limitations

- Flash invalidation trick only works against very old routers
- Guessing previous pointer values is usually infeasible

Overcoming The Obstacles

- **Disassembly Ninjitsu**
- **Lots Of Hard Work**
- **Cisco Helps Us Out Some**
 - Built in debugger (sort of)
 - show mem commands
 - show context
 - Forced core dumps
 - debug all
- **Finding The IOS Version**
 - CDP
 - SNMP
 - Read Only Buffer Overflows

Getting IOS In A Disassembler

- **Use A Core Dump Image**
 - This will show you memory contents of the system during runtime
- **Decompress The Firmware Image**
 - Stuffit expander
 - WinRar
 - Fixup the ELF header
- **Be Prepared for IDA To Run Dog Slow**

■ Stack Overflows

- These just work if you can find them

■ Heap Overflows

- We need a pointer exchange
- Its best if we can overwrite something other than heap linkage
- Hijack any number of callbacks

■ Using Heap Linkage

- We can't overflow past the next pointer
- Maybe we could use FX's uncontrolled pointer exchange method for something useful

Easy Heap Overflows

■ **Overwrite Linked List In Same Chunk**

- Doesn't clobber heap chunks
- Take control with pointer exchange
- Easy and reliable, but somewhat rare

■ **Overwrite Linked List In Another Chunk**

- We are racing against check heaps
- Our chunk must not be freed
- We are racing check heaps
- Very hard in practice unless we can deal with check heaps

Hard Heap Overflows

■ Racing Check Heaps

- We have between a few seconds and a minute to get execution or we'll be busted by check heaps
- Sometimes we can trigger the unlink and force us to win the race
- Sometimes we can't

■ Lets Kick Check Heaps In The Nuts

- What if we could make check heaps go away
- What if we could not let the router crash
- This would greatly increase our chances of success
- Lets take a look at how the system crashes

Inside The abort() Routine

```
stwu    sp, var_18(sp)
mflr   r0
stmw   r29, 0x18+var_C(sp)
stw    r0, 0x18+arg_4(sp)
lis    r9, (crashing_already_ >> 16)
lwz    r0, (crashing_already_ & 0xFFFF)(r9)
cmpwi  r0, 0
bne    loc_80493D18    # return
```

I Never Liked Check Heaps Anyways

- **Use Uncontrolled Pointer Exchange To Trick System Into Thinking It Is Already Crashing**
 - Router can no longer crash synchronously
 - Check heaps will eventually be killed due to CPU hog watchdog
- **This Buys You A Few Minutes**
 - The system will still eventually crash on an unhandled exception anyways
- **This Gives The Potential To Exploit Arbitrary Heap Overflows**
 - After check heaps is dead it may be possible to use uncontrolled pointer exchange to get execution
 - You can now guess previous pointer values and the system can't crash

Building The Shellcode

- **Memory Allocation**
 - malloc
- **Process Management**
 - CreateThread
 - exit
- **TTY Management**
 - allocateTTY
 - Setting up a tty
- **Sockets (well, sort of)**
 - TCBCreate
 - Connect

Finding malloc()

```
9      li      r3, 0xC
4      bl      malloc
8      mr.     r4, r3
C      beq     loc_80E44D0C
9      stw    r28, 4(r4)
4      stw    r27, 8(r4)
8      addi   r3, r30, 0xCC
C      bl      sub_8049CA30
9      lwz    r9, 0xC0(r30)
4      stbx   r28, r9, r27
8      lwz    r9, 0xC0(r30)
C      stbx   r28, r9, r28
9      lwz    r9, 0xBC(r30)
4      stbx   r27, r9, r28
8      b      loc_80E44E08
```

```
C # -----
C
C loc_80E44D0C:      # CODE XREF: sub_80E44C18+C4↑j
C                   lis      r3, ((aAddressAddFail+0x10000) >> 16) # 0x821289CC # "Address add failed
9                   subi   r3, r3, 0x7634 # aAddressAddFail # "Address add failed due to no memory"
4                   bl      printf_
8                   b      loc_80E44DC4
C # -----
```

Finding CreateThread()

```
stwu    sp, var_10(sp)
mflr   r0
stw    r31, 0x10+var_4(sp)
stw    r0, 0x10+arg_4(sp)
lis    r31, (dword_82F00400 >> 16)
lwz    r0, (dword_82F00400 & 0xFFFF)(r31)
cmpwi  r0, 0xFFFF
bne    loc_80610F70
lis    r3, (DHCPD_Receive >> 16) # 0x80610CB0
addi   r3, r3, (DHCPD_Receive & 0xFFFF) # (DHCPD_Receive & 0xFFFF)
lis    r4, ((aDhcpdReceive+0x10000) >> 16) # 0x81E7F36C # "DHCPD Receive"
subi   r4, r4, 0xC94 # aDhcpdReceive # "DHCPD Receive"
li     r5, 0x1770
li     r6, 3
bl     createThread
```

Using CreateThread()

```
void *CreateThread(void *entryPoint,  
                  char *name,  
                  int something,  
                  int dunno);
```

Finding exit()

```
#-----  
loc_80484098:                                # CODE XREF: kill_process+1C↑j  
    lbz     r0, 0x78(r31)  
    cmpwi  r0, 0  
    beq    loc_804840C4  
    lis    r9, ((kern_err_msg+0x10000) >> 16)  
    lwz    r0, (kern_err_msg & 0xFFFF)(r9)  
    lis    r3, ((off_81DFFE44+0x10000) >> 16) # 0x81DFFE44  
    subi   r3, r3, 0x1BC # off_81DFFE44  
    lwz    r4, 0xD8(r31)  
    lwz    r5, 0x88(r31)  
    mtlr   r0  
    blrl  
  
loc_804840C4:                                # CODE XREF: kill_process+48↑j  
    lis    r9, ((CURRENT+0x10000) >> 16) # this current process?  
    lwz    r0, (CURRENT & 0xFFFF)(r9) # this current process?  
    cmpw   r31, r0  
    bne    loc_804840DC  
    bl     exit  
    b     loc_80484148 # return
```


An Example Of TTY Creation

```
loc_80B18A80:                                # CODE XREF: sub_80B1863C+424↑j
      li      r3, 0x17
      lis    r4, ((pad_io+0x10000) >> 16) # 0x82F0E3B8
      subi   r4, r4, 0x1C48 # pad_io
      bl     get_ttygroup # not entirely sure what this is, returns something
                                     # to pass in to allocatetty
      mr     r30, r3
      ble   loc_80B18AAC
      mr     r3, r30
      li    r4, 1
      bl    allocateTTY
      mr     r31, r3
      bne   loc_80B18AC8
```

Using TTY Routines

ttygroup

```
*getTTYGroup(int twentyOne, io_t *ioStruct);
```

tty_t

```
*allocateTTY(ttygroup *group, int one);
```

We Need A Socket

- **Too Bad, This Is Not Unix, Its Not Even Close**

- Actually they do have BSD style sockets, they are just never used and are not helpful to us

- **TCB's**

- I don't know what this stands for, and neither did the people at Cisco I spoke with
- This is the socket like thing we have to use
- They seem comparable to sockets, but work in an asynchronous way

Lets See How TCB's Are Used

```
                                # CODE XREF: sub_805B7434+344fj
li      r3, 0
addi   r4, sp, 0x140+var_30
li     r5, 0
bl     | tcp_create      # creates some structure used in their socket like
                                # thing...
```

Another Example

```
loc_80558C58:                                # CODE XREF: tcp_create_connect+7C↑j
                                                # tcp_create_connect+98↑j
                                                # tcb
                                                # address
                                                # port
                                                # flag/mode
                                                # backend of connect functions
    mr      r3, r31
    mr      r4, r27
    mr      r5, r25
    mr      r6, r26
    bl      _tcp_connect
    stw     r3, 0(r29)
    cmpwi   r3, 1
    mr      r3, r31
    beq     loc_80558C84    # return

loc_80558C7C:                                # CODE XREF: tcp_create_connect+A0↑j
    bl      tcp_close    # something to do with closing tcb's
```

TCB

```
*tcp_create_connect1 (int zero,  
                      short remotePort,  
                      sockaddr *remoteAddr,  
                      short localPort,  
                      sockaddr *localAddr,  
                      int *error,  
                      int zero);
```

A Dead Process Tells No Tales

■ **Lets Cover Our Tracks**

- We could flush the logs
- We could modify the log strings on the heap
- We could sabotage the logging functions

■ **Or We Could Just Kill The Logger Daemon**

- Some messages still appear on reboot, but only to console as best I can tell

Finding Kill

```
                .byte    0
aNotdead:      .string  "NOTDEAD"      # DATA XREF: RAM:81DFFE48↑to
off_81DFFE88:  .long    aSys          # DATA XREF: sub_80482744+8C↑to
                                   # sub_80482744+90↑to ...
                                   # "SYS"
                .long    aNoProcess  # "NOPROCESS"
                .long    aNoSuchProcessD_0 # "No such process %d"
```


Shellcode Check List

1. **Get Execution**
2. **Clean Up What We Broke**
3. **Spawn Process**
4. **Allocate And Setup TTY**
5. **Make Connect-Back TCB**
6. **Start Shell**
7. **Kill Logger Process**
8. **Exit Initial Process**
9. **World Domination**

Is This The End Of The World

■ **Yes And No (Mostly No)**

- Cisco is working on this
- Keep your firmware images up to date and you will probably be fine
- Because you have to have different offset for different firmware versions worms would be very difficult to make

■ **But Then Again**

- Stack overflows do not need to know router versions to gain execution
- Up coming versions of IOS use “virtual processes” this means that offsets will be static between firmware versions

Questions?

Questions?